FORMAL SPECIFICATIONS OF KVM/370 KERNEL AND TRUSTED PROCESSES

B. GOLD
D. THOMPSON

21 MAY 1978

TM-6062/101/00

339 100

3410

21 May 1978 System Development Corporation
TM-6062/101/00

Contract MDA903-76-C-0260

Table of Contents

INTRODUCTION

This document contains the formal specification of the kernel and trusted processes of the Kernelized VM operating system (KVM/370). The specification utilizes the specification language INA JO, a proprietary product of System Development Corporation. INA JO is described in TM-6021/000/00, January 1978. This document assumes familiarity with INA JO.

The specification is divided into two main parts. Chapter two contains the specification of the kernel, and is split into nineteen sections corresponding to the main areas of functionality of the kernel. For the most part, little or no explanation of the kernel specification is provided; a prose description of each kernel function is contained in "Preliminary Design of Security Kernel for KVM/370", TM-5855/200/00, October 1977. Chapter three contains the specifications of the trusted processes. Each of the five trusted process descriptions has two subsections. The first subsection for each trusted process contains a prose description of the process, detailing its function. The second subsection contains the INA JO specification of the process.

2.1: KVM Security Policy

This section describes the security policy implemented by KVM/370. A security level is defined as an ordered pair (L,C), where L is one of (Unclassified, Confidential, Secret, TopSecret), and C is a (possibly empty) set of special access categories (e.g., Crypto, Atomic Energy). A function 'Dominates' is defined which represents the concept 'greater than or equal to' as applied to security levels. The types of access which can be granted and the concept of an access history are defined. The access history is an answer to the question "has subject S ever had read or write access to object O?"

The security criterion for the system is defined in two parts. The first is the non-discretionary policy and requires that:

(1) a subject's clearance must dominate the classification of any object to be read; and

(2) a subject's security level must exactly equal that of any object to be written.

The second part consists of the discretionary need-to-know controls, implemented as access control lists which specify whether or not a given object is protected against a particular type of access, and which subjects may have the specified access to the object.

```
specification KVM←Security←Kernel

module TopLevel

          /☆ Definition of Security Level and related Functions ☆/

type HiarchyLvl = (Unclassified, Confidential, Secret, TopSecret),
          SpecialProtectionCompartment,
          Categories = set of SpecialProtectionCompartment,
          Class = HiarchyLvl >< Categories

constant LowLevel = L"H:HiarchyLvl(A"J:HiarchyLvl(H <= J)).1 /☆Unclassified☆/,
          HighLevel = L"H:HiarchyLvl(A"J:HiarchyLvl(H >= J)).1 /☆TopSecret☆/,
          SystemLow = (LowLevel, empty),
          SystemHigh = (HighLevel, S"C:SpecialProtectionCompartment (true) )

constant DOMINATES(dominator: Class, dominated: Class) =
          (dominator.1 >= dominated.1 & dominator.2 >>= dominated.2)
          MergeUp(A: Class, B: Class) ==
                   (  (A.1 > B.1 => A.1 <> B.1),  A.2 || B.2  )
          MergeDown(A: Class, B: Class) ==
                   (  (A.1 < B.1 => A.1 <> B.1),  A.2 && B.2  )


          /☆ subjects and objects ☆/

type Object, Subject < Object, AccessType

constant None, ReadOnly, ReadWrite :AccessType
distinct (None, ReadOnly, ReadWrite)

variable GrantedAccess (Subject, Object): AccessType,
           SecurityLevel (Object):  Class,
           Trusted (Subject): boolean

          /☆ access history ☆/
variable ReadAccessObtained(Subject, Object),
          WriteAccessObtained(Subject, Object): boolean

          /☆ non-discretionary security ☆/

criterion A"S:Subject, O:Object (TRUSTED(S) |
    /☆ "Security Property" ☆/
          (ReadAccessObtained(S,O) ->
                   Dominates(SecurityLevel(S), SecurityLevel(O)) )
    /☆ "☆-Property" ☆/
          & (WriteAccessObtained(S,O) ->
                   SecurityLevel(S) = SecurityLevel(O)) )

    /☆ Discretionary Need-to-Know Access Controls ☆/

type      AccessList = structure of
                   (Read  = set of Subject,
                    Write = set of Subject,
                    Protr  = boolean,
                    Protw  = boolean )

variable DirectoryAccess(Subject,Object):AccessType,
         ACL(Object): AccessList

criterion A"S:Subject, O:Object
        ( Trusted(S) |
    /* read access protection */
          (ReadAccessObtained(S,O) & ACL(O).Protr ->
                   S<:(ACL(O).Read || ACL(O).Write))
    /* write access protection */
          & ( WriteAccessObtained(S,O) & ACL(O).Protw -> S<:ACL(O).Write))
    /* The input parameter "R" used in these specifications corresponds
     to the parameter "REQUESTER" in the informal specifications
according to the following assertion,
which holds for all Kernel calls
     E1"R: KProcBlok (R<:ProcessList & R.Procname = Requestor)

R is the KProcBlok uniquely defined by the above */
```

2.2: Machine Specifications

This section contains the machine-defined structures and data. The reader interested in more details about these machine-dependencies is referred to "IBM System/370 Principles of Operations", GA22-7000-5. International Business Machines Corporation, August 1976.

```
   /* SYSTEM/370 MACHINE DEFINED STRUCTURES AND DATA */

        /* Address Translation (DAT) Structures */
type Byte
type PageOfBytes = T"L: list of Byte (C"L = PageSize*1024)

            /* real page frame address >< validity */
type PageTableEntry = VPage >< boolean

        /* For data structure purposes, page tables are thought of as
                containing the addresses of the pages they refer to,
                and segment tables as containing the page tables */
type PTEList = list of PageTableEntry,
        PageTable = T"P: PTEList (C"P = PagesPerSegment),
        SegmentTableEntry = PageTable >< boolean,
        STEList = list of SegmentTableEntry

        /* storage keys */
type ProtectionKey,
        KeyInStorage = structure of
            (Key = ProtectionKey,
             NoFetch = boolean,
             Refer = boolean, Change = boolean)
constant Key0: ProtectionKey

        /* Addressing Technique */
type VPage = T"I: integer (0 <= I & I < 16*1024/PageSize),
     ByteOffset = T"I: integer (0 <= I & I < PageSize*1024),
     Address370 = VPage >< ByteOffset
```

2.3: Paging Parameters

This section defines the parameters and data types controlling paging and address space manipulations.

```
    /* paging system constants */

constant PageSize = 2 /*2K=2048*/,  SegmentSize = 64 /*64K=65536*/,
         PagesPerSegment = SegmentSize/PageSize /*32*/,
         PageTableEntrySize = 2 /*Halfword=2 bytes*/, PageTableHeaderSize = 8,
         SegmentTableEntrySize = 4, SegmentTableHeaderSize = 0,
         SwapTableEntrySize = 8, SwapTableHeaderSize = 8,
         PageTableSize = PageTableHeaderSize
                 + PagesPerSegment*PageTableEntrySize,
         SwapTableSize = SwapTableHeaderSize
                 + PagesPerSegment * SwapTableEntrySize,
         RoundSegments(X: VPage) = (X + PagesPerSegment - 1)/PagesPerSegment
   /* type definitions for virtual address structure */
type Lock = T"I:integer ( I >= 0)
type Status = (NKCPData, Free, EmptyPage, ReadOnly, ReadWrite, IN, OUT)
type PageFrameName, PageSlotName, SlotAddress, AddressSpaceDesignator

constant P0: PageFrameName /* "name" of an NKCP's page 0 */

type PageFrame = structure of
          (Addr = VPage,
           Status = (NKCPData, Free, EmptyPage, ReadOnly, ReadWrite),
          /* the following fields are meaningful only if Status ~= Free */
           Owner = ProcessName, VirtualName = PageFrameName,
           TLOCK = boolean,
           Intransit = (IN, OUT, FREE),
          /* following meaningful only if status = ReadOnly or ReadWrite */
           RealKey = KeyInStorage,
           ILOCK = Lock, OLOCK = Lock, ULOCK = boolean,
           VirtAddr = VirtualPage, Contents = PageOfBytes),
     PageSlot = structure of
          (Addr = SlotAddress,
           Status = T"(Free, EmptyPage, ReadOnly, ReadWrite),
           /* following fields meaningful only if Status ~= Free */
           Owner = ProcessName, Sname = PageSlotName,
           /* following field meaningful
                   only if Status = ReadOnly or ReadWrite */
           Contents = PageOfBytes),
     VirtualPageName = structure of
          (VMname = VirtualMachineName,
           A = AddressSpaceDesignator,
           Vaddr = VPage),
     VirtualPage = structure of
          (Status = T"(ReadOnly, ReadWrite, EmptyPage),
           /* following fields meaningful only if Status ~= EmptyPage */
          VSlot = PageSlotName,
          SavedKey = KeyInStorage, Refer = boolean, Change = boolean,
          Contents = PageOfBytes,
          Valid = boolean,
          /* following meaningful only if Valid = true */
          Raddr = VPage /* real page frame, if any */,
          VaddrList = set of VirtualPageName )

variable RealAddress(ASpace, VPage): PageTableEntry,
        VirtPage(ASpace, VPage): VirtualPage

variable SharedSegmentList: set of PageTable,
        RealPages: set of PageFrame,
        RealSlots: set of PageSlot

invariant A"P1,P2: PageFrame (P1<:RealPages & P2<:RealPages
                -> (P1.VirtAddr.Valid -> P1.VirtAddr.Raddr = P1.Addr)
        & ((P1.Addr = P2.Addr |
                P1.Owner = P2.Owner & P1.VirtualName = P2.VirtualName
                        & P1.Status ~= Free & P2.Status ~= Free
                | P1.VirtAddr.Valid & P2.VirtAddr.Valid & E"V: VirtualPageName
                                (V<:P1.VirtAddr.VaddrList &
                                 V<:P2.VirtAddr.VaddrList))
            -> P1 = P2))
        & S1,S2: PageSlot (S1<:RealSlots & S2<:RealSlots
                        & (S1.Addr = S2.Addr
                          | S1.Status ~= Free & S2.Status ~= Free
                                & S1.Owner = S2.Owner & S1.Sname = S2.Sname)
                -> S1 = S2)
```

2.4: Legality Checking Macros

This section defines several macros which are heavily utilized throughout the remainder of the kernel specification (all of Chapter two).

```
macro CheckVM(R: KProcBlok, V: VirtualMachineName) ==
        V = SYSTEM
        | LET"K = U"K: KVMblock (K<:KVMtable & K.VMname = V)
          & K.Owner = R.ProcName

macro CheckASpace(R: KProcBlok,
                  V: VirtualMachineName,
                  A: AddressSpaceDesignator) ==
        CheckVM(R,V)
        & LET"AS = (V = SYSTEM => R.AddressSpace
                   <> AddressSpace(K, A)  )
        & AS.Exists

define CheckVPage(S: ASpace, P: VPage) ==
        S.Exists & P < S.NumberOfPages

macro CheckMPage(R: KProcBlok,
                V: VirtualMachineName,
                A: AddressSpaceDesignator,
                P: VPage) ==
        CheckASpace(R, V, A) & CheckVPage(AS, P)

macro CheckFrame(R: KProcBlok, Pframe: PageFrameName) ==
        LET"PF = U"P:PageFrame (P.Owner = R.ProcName & P.VirtualName = Pframe)

macro CheckSlot (R: KProcBlok, Slot: PageSlotName) ==
        LET"PS = U"S:PageSlot (S.Owner = R.ProcName & S.SName = Slot)
```

2.5: Address Space Definitions and Functions

This section describes virtual address spaces and the functions that create and destroy them.

```
   /* Address Space Definitions */
type ASpace = structure of
          ( /* first two fields uniquely identify this ASpace */
           A = AddressSpaceDesignator, Owner = Subject,
           Exists = boolean,
            /* the following are meaningless if ~Exists */
           PurgeTLBNeeded = boolean,
           NumberOfPages = VPage,
           Addresses = T"S:STEList (C"S > 0 & C"S <= 16*1024/SegmentSize))

variable AddressSpace(KVMblock, AddressSpaceDesignator): ASpace

variable Spaces: set of ASpace

constant NullSpace(AS:AddressSpace):boolean ==
          A" S:SegmentTableEntry, P:PageTableEntry
               (S<:AS -> ~S.2 | (P<:S.1 -> ~P.2) )
          & A"i:VPage (i < A.NumberOfPages -> VirtPage(AS,i).Status = EmptyPage)

invariant A"S1, S2: ASpace, K:KVMblock, A:AddressSpaceDesignator, P: KProcBlok
                    (S1<:Spaces & S2<:Spaces
                                    & K<:KVMtable & P<:ProcesssList
                      -> S1.Exists & S2.Exists
                            & (S1.Owner = S2.Owner & S1.A = S2.A -> S1 = S2)
                            & (P.AddressSpace = S1 => S1.A = 0
                                    & S1.Owner = P.ProcessName
                               <> AddressSpace(K,A) = S1
                                    -> S1.A = A & S1.Owner = K.VMname)
                            & E"K1: KVMblock,
                               P1: KProcBlok, A1: AddressSpaceDesignator
                                  (P1.AddressSpace = S1 |
                                   AddressSpace(K1,A1) = S1))
```

```
transform CreateAddressSpace (VM: VirtualMachineName,
                              Size: VPage,
                              A: AddressSpaceDesignator,
                              R: KProcBlok)

refcond R<:ProcessList & R.Proctype = NKCP
        & VM ~= SYSTEM & CheckVM(R, VM)
        & ~AddressSpace(K,A).Exists

effect ( AreaOK(R.ProcName, (SegmentTableHeaderSize +
                    RoundSegments(Size) *
                         (SegmentTableEntrySize
                              + PageTableSize + SwapTableSize)))
                => ReturnCode(R,0)
                    & E"S:ASpace (N"AddressSpace(K,A) = S &
                          NullSpace(S) & S.NumberOfPages = Size
                                  & N"Spaces = Spaces || S"(S)
                                  & AdjustQuota(R.ProcName,
                                             SegmentTableHeaderSize
                  + RoundSegments(Size) *
                         (SegmentTableEntrySize
                                  + PageTableSize + SwapTableSize) ) )
            <> ReturnCode(R,1) & N"Spaces = Spaces
                  & N"K = K & AdjustQuota(R.ProcName,0) )
```

```
transform Destroy AddressSpace(VM: VirtualMachineName,
                               A: AddressSpaceDesignator,
                               R: KProcBlok)

refcond R<:ProcessList & R.ProcType = NKCP
        & CheckAspace(R,VM,A)
        & NullSpace(AS)

effect N"Spaces = Spaces ~~ S"(AS) & ~N"AS.Exists
        & AdjustQuota(R.ProcName, -(SegmentTableHeaderSize +
                     RoundSegments(Size)*
                     (SegmentTableEntrySize+PageTableSize+SwapTableSize)))
```

2.6: Create/Destroy Process

This section describes KVM's definition of a process and the functions that create and destroy processes.

```
type ProcessName = Class,
     KProcBlok = structure of
         (ProcName = ProcessName,
          Proctype = (NKCP, TRUSTED, AUDITED),
          MessageQueue = list of MsgBlok,
          MessagePending = boolean, MessagePendingReceived = boolean,
          IOInterrupts = set of IOintBlok, IOInterruptPending = boolean,
          ClockCompInterruptPending = boolean,
          TrackingQuantum = boolean,
          QuantumEnded = boolean,
          CPUTimerInterruptPending = boolean,
          RemainingQuantum = SignedDoubleword, CPUtimer = SignedDoubleword,
          IntvlTimer = integer, IntvlTimerInterruptPending = boolean,
          Internals = IntState,
          AddressSpace = ASpace)

variable ProcessList: set of KProcBlok
```

```
transform CreateProcess (Asize,
                         Csize: VPage,
                         Code: list of PageFrameName,
                         Page0: PageFrameName,
                         Process: ProcessName,
                         R: KProcBlok)

refcond R.ProcName = INITIATOR &
                 ProcessCount < MaxProcessCount /* integer constant */
         & CSize < ASize & CSize = C"Code
         & A"I: VPage (   I <= CSize -> E1"PF:PageFrame (PF.Owner = INITIATOR
                                                         & PF.VirtualName =
                                                           (Code;Page0).(I+1) )
                            & E1"PF:PageFrame (PF.Owner = INITIATOR
                                    & PF.VirtualName = (Code;Page0).(I+1)
                                    & PF.Status = (I=0 => EmptyPage
                                                   <> ReadWrite)))

effect (     ~SlotsOK(ASize-CSize) => ReturnCode(R,1)
                  & N"ProcessList = ProcessList & N"Spaces = Spaces
                  & A"PF:PageFrame (N"PF = PF)
         <> ~AreaOK(INITIATOR, (KProcBlokSize + ASpaceSize(ASize)) ) =>
                  ReturnCode(R,2)
                  & N"ProcessList = ProcessList & N"Spaces = Spaces
                  & A"PF:PageFrame (N"PF = PF)
         <> ReturnCode(R,0) & AdjustQuota(INITIATOR,KProcBlokSize
                                                         + ASpaceSize(ASize))
           & E"P:KProcBlok, S:ASpace, PF:PageFrame (N"Spaces = Spaces ↑↑ S"(S)
                  & S.NumberOfPages = Asize
                  & N"ProcessList = ProcessList || S"(P) & P.ProcName = Process
                  & PF.Owner = INITIATOR & PF.VirtualName = Page0
                  & N"PF.Owner = Process
                  & N"PF.VirtualName = P0 /* PageFrameName Constant */
                  & P.AddressSpace = S & NoInterruptsPending(P)
                  & S.PurgeTLBneeded
                  & A"I:VPage ( (I<=CSize -> RealAddress(S,I).2
                                   & E"PF:PageFrame
         (PF.Owner = Initiator & PF.VirtualName = (Code;Page0).(I+1)
          & PF.Addr = RealAddress(S,I).1 )
                                   & (I > CSize & I < ASize ->
                                            ~RealAddress(S,I).2))))
```

```
transform DestroyProcess(Process: ProcessName,
                         R: KProcBlok)

refcond R.ProcName = INITIATOR & R<:ProcessList
        & LET"P = U"P: KProcBlok (P<:ProcessList & P.ProcName = Process)
        & A"D:IODevice(GrantedAccess(P.ProcName, D.Address) = None)
        & A"M:MiniDiskName (GrantedAccess(P.ProcName, M) = None)
        & A"S:SharedSegmentName (GrantedAccess(P.ProcName, S) = None)

effect E"S:ASpace (S<:Spaces & P.AddressSpace = S
                   & ( A"I:VPage (I < S.NumberOfPages
                           & RealAddress(S,I).2 -> A"PF: PageFrame
                               (PF.Addr = RealAddress(S,I).1 ->
                                       PF.Status = ReadOnly))
                       => N"Spaces = Spaces ~~ S"(S)
                              & N"ProcessList = ProcessList ~~ S"(P)
                         & ReturnCode(R,0)
                         & AdjustQuota(INITIATOR, - (KProcBlokSize
                              +ASpaceSize(S.NumberOfPages))) )
                   <> ReturnCode(R,1) )
```

2.7: Create/Destroy VM

This section describes KVM's virtual machines and the functions that create and destroy them.

```
type VirtualMachineName,
        KVMblock = structure of
                (VMname = VirtualMachineName, Owner = ProcessName)

constant SYSTEM: VirtualMachineName
variable KVMTable: set of KVMblock
```

```
transform CreateVM(VM: VirtualMachineName,
                   NCKPName: ProcessName,
                   R: KProcBlok)

refcond R.ProcName = INITIATOR
        & E1"P: KProcBlok (P<:ProcessList & P.ProcName = NKCPName
                           & P.ProcType = NKCP)

effect (    E"K:KVMblock (K<:KVMtable & K.VMName = VM)
               => ReturnCode(R,1)
        <> A"K:KVMblock (K<:KVMtable -> K.VMName ~= VM)
               => E1"K:KVMblock (K.VMName = VM & K.Owner = NKCPName
                     & A"A:AddressSpaceDesignator(
                           ~AddressSpace(K,A).Exists )
                     & N"KVMTable = KVMTable || S"(K) )  )
```

```
transform DestroyVM (VM:VirtualMachineName,
                     R: KProcBlok)

refcond LET"K = U"K: KVMblock (K<:KVMtable & K.VMName = VM)
                & A"A:AddressSpaceDesignator (Nullspace(AddressSpace(K,A)))
        & (R.ProcName = INITIATOR | R.ProcName = K.Owner)

effect N"KVMtable = KVMtable ~~ S"(K)
```

2.8: Grant/Revoke Functions

This section describes real IO devices and shared (memory) segments, and defines the functions that grant and revoke access to them.

```
type HeadNumber = T"H:integer (H >= 0 & H <= 50),
     CylinderNumber = T"C:integer (C >= 0 & C <= 10000),

type ObjectName = structure of
        (KindOfObject = (SharedSegment, HorizontalMinidisk,
                         VerticalMinidisk, WholeDevice),
         /* Pages meaningful only if KindOfObject = SharedSegment,
                Vol only if KindOfObject = Horizontal or VerticalMinidisk
                Slice only if KindOfObject = HorizontalMinidisk,
                Start and Finis only if KindOfObject = VerticalMinidisk,
                Dev only if KindOfObject = WholeDevice */
         Pages = list of PageFrameName,
         Vol = RealVolumeId,
         Slice = HeadNumber,
         Start = CylinderNumber,
         Finis = CylinderNumber,
         Dev = DeviceAddress)
constant NullDevice: DeviceAddress
type OwnedVolume = structure of
        (Name = RealVolumeId,
         Device = DeviceAddress)

variable OwnedList: set of OwnedVolume,
        DeviceList: set of RealDevice,
        SharedVolumeDevices: set of DeviceAddress

constant Devtype(UnitType): DeviceDescription

define DeviceExists(D: DeviceAddress) == LET"Dv = U"Dv: RealDevice
                (Dv.Addr = D & Dv<:DeviceList),
        VolumeMounted(V: RealVOlumeId) ==
                LET"VOL = U"VOL: OwnedVolums (
                        VOL.Name = V & VOL<:OwnedList)
                & VOL.Device ~= NoDevice
```

```
type Grant = structure of
        (Access = set of (ReadOnly, ReadWrite),
         Class = (DASD, Terminal, Tape, UnitRecord),
         Dedicate = boolean,
         /* following meaningful only if Class = DASD.
                Tracks meaningful only if Horizontal = true,
                Cyls only if Vertical = true                    */
         Horizontal = boolean,
         Vertical = boolean,
         Tracks = set of HeadNumber >< Access,
         Cyls = set of CylinderNumber >< Access )

constant NoAccess: Grant
```

```
transform GrantAccess(Object: ObjectName,
                      Acc: (ReadOnly, ReadWrite),
                      P,
                      R: KProcBlok)

refcond TRUSTED(R.ProcNAME) & R<:ProcessList & P<:ProcessList
        & (Object.KindOfObject = SharedSegment =>
                 R.ProcName = INITIATOR
                 & C"Object.Pages = PagesPerSegment
                 & A"Pframe: PageFrameName (Pframe<:Object.Pages
                      -> CheckFrame(R, Pframe)
                           & PF.Status<:(ReadOnly, ReadWrite))
           <> Object.KindOfObject = HorizontalMinidisk =>
                 VolumeMounted(Object.Vol)
                      & Vol.Name = Object.Vol & Vol.Division = Horizontal
                      & Slice >= 0
                      & Slice < Devtype(V.DeviceType).TracksPerCylinder
           <> Object.KindOfObject = VerticalMinidisk =>
                      VolumeMounted(Object.Vol)
                      & Vol.Division = Vertical
                      & Start >= 0 & Start <= Finis
                      & Finis < Devtype(Vol.DeviceType).NumCyl
           <> Object.KindOfObject = WholeDevice =>
                      DeviceExists(Object.Dev)
                         & Object.Dev ~<:SharedVolumeDevices )

effect (Object.KindOfObject = SharedSegment => E1"PT: PageTable
                      (PT<:N"SharedSegmentList
                       & A"i: integer (i > 0 & i <= PagesPerSegment
                            -> CheckFrame(R, Object.Pages.i)
                               & PT.i = PF)
                       & PT<:N"SharedSegmentList(P)
                       & PT.Name = Object.Name)
        <> Object.KindOfObject = WholeDevice =>
                 LET"G = GrantedAccess(P.ProcName, Object.Dev)
                 & N"G.Access = G.Access || S"(Acc)
                 & N"G.Dedicate = true
                 & N"G.Class = DeviceType(Object.Dev).Class
        <> VolumeMounted(Object.Vol)
           & LET"G = GrantedAccess(P.ProcName, VOL.Device)
           & N"G.Access = G.Access || S"(Acc)
           & N"G.Class = DASD & ~N"G.Dedicate
           & (Object.KindOfObject = HorizontalMinidisk =>
                      N"G.Horizontal & ~N"G.Vertical
                      & N"G.Tracks = G.Tracks || S"(Object.Slice, Acc)
               Object.KindOfObject = VerticalMinidisk =>
                      N"G.Vertical & ~N"G.Horizontal
                      & N"G.Cyls = G.Cyls ||
                            S"Z: CylinderNumber >< (ReadOnly, ReadWrite)
                                  (Z.1 >= Start & Z.1 <= Finis
                                   & Z.2 = Acc) ))
```

```
transform ReleaseDevice(Device: DeviceAddress,
                        P,
                        R: KProcBlok)

refcond R.ProcType = Trusted
        & GrantedAccess (P.ProcName, Device) ~= NoAccess
        & A"LB: LockedBox (LB<:LockedBoxList
                           & LB.Notify = P.ProcName -> LB.Dev ~= Device)

effect N"GrantedAccess(P.ProcName, Device) = NoAccess
```

2.9: Attach/Detach Page Functions

This section defines the functions which place and remove virtual pages in virtual address spaces.

```
transform AttachPage(Pframe: PageFrameName,
                     Addr:  VirtualPageName
                     R: KProcBlok)

refcond CheckFrame(R, Pframe)
          & CheckMPage(R, Addr.VMname, Addr.A, Addr.Vaddr)
          & LET VP = VirtPage(A,Addr.Vaddr) & LET Ad = RealAddress(A,Addr.Vaddr)
          & PF.ILOCK= 0 & PF.Status<:S"(EmptyPage,ReadWrite)
          & (Ad.2 -> U"P: PageFrame (P.Addr = Ad.1).Status = ReadWrite)

effect     N"Ad.1 = PF.Addr & N"Ad.2
          & N"PF.Status = ReadWrite & N"PF.TLOCK
          & N"PF.RealKey.Key = (VP.Status = EmptyPage => Key0
                                 <> VP.SavedKey.Key)
          & N"PF.RealKey.NoFetch = (VP.SavedKey.NoFetch & VP.Status ~= EmptyPage)
          & (PF.VirtAddr.Valid -> ~N"PF.RealKey.Refer & ~N"PF.RealKey.Change
                   & ~N"VP.Change  & ~N"VP.Refer )
          & N"VP.Status = ReadWrite
          & Ad.2 -> N"A.PurgeTLBNeeded
          & PF.Status = EmptyPage -> N"PF.Contents = 0
          & N"PF.VirtAddr = N"VP & N"VP.VaddrList = VP.VaddrList || S"(Addr)
          &N"VP.Valid
```

```
transform DetachPage(Addr: VirtualPageName,
                        R: KProcBlok)

refcond CheckASpace(R,Addr.VMname,Addr.A)
        & LET AD = RealAddress(A,Addr.Vaddr)
        & AD.2

effect N"A.PurgeTLBNeeded & ~N"AD.2
        & N"VP.VirtList = VP.VirtList ~~ S"(Addr)
        & (N"VP.VirtList = empty -> ~N"VP.Valid)
```

2.10: Swapping Functions

This section describes the functions overseeing the movement of real pages between real memory and direct-access storage devices. The system protects against mis-seeks by associating a key with each record on a disk track. This key is an encrypted address of the record. After locating the record (via SEARCHID), the key is verified to be the particular one desired (via SEARCHKEY). The key is not visible to the user nor to untrusted control software.

EnqueueSwap is not described in the preliminary design documentation provided in TM-5855/200/00. This function is a common routine used by SwapIn and SwapOut. It enters swap requests into the queue of outstanding IO requests.

```
type SearchType < CCWOpCode, SlotKey,
        SwapCheck = SearchType >< SlotKey >< integer,
        SwapBlok = structure of
                 (ReqType = (IN, OUT),
                  Active = boolean,
                  Frame = VPage,
                  Slot = SlotAddress,
                  Notify = ProcessName,
                  ID = RequestName,
                  Check = SwapCheck),
        SwapReq = list of SwapBlok

constant Clear,Nochecks:SwapCheck, NullSwap: SwapBlok
        RecNum(SlotAddress), TrackAddress(SlotAddress): integer

variable SwapQueue: set of SwapReq

invariant A"SR: SwapReq((SR<:SwapQueue
                 -> A"B: SwapBlok (B<:SR -> B.Slot.1 = SR.1.Slot.1))
                 & A"T: SwapReq (T<:SwapQueue
                         & T.1.Slot.1 = SR.1.Slot.1 -> T = SR))
```

```
transform EnqueueSwap(PF: PageFrame,
                      Direction: (IN,OUT)
                      Slot: PageSlot,
                      Chk: SwapCheck,
                      Notify: KProcBlok,
                      ID: RequestName)

refcond PF.ILOCK = 0 & PF.OLOCK = 0 & ~PF.ULOCK
        & PF.Intransit<:S"(Free,Direction)

effect  (Direction = OUT -> SetSlot(PF,Slot) )
        & N"PF.Intransit = Direction
        & E"SR: SwapReg, B: SwapBlok (B<:SR & SR<:N"SwapQueue
                      & B.ReqType = Direction
                      & B.Frame = PF.Addr & B.Slot = Slot.Addr
                      & B.Check = Chk
                      & B.Notify = Notify.ProcName & B.ID = ID)

   /* future SWAP Daemon effect
        A" SR: SwapReq, B: SwapBlok (SR<:N"SwapQueue & B<:SR
                      -> (B.ID ~= ID | B.Notify ~= Notify.ProcName) )
        & (Direction = OUT => N"Slot.Contents = PF.Contents
           <> Direction =IN => N"PF.Contents = Slot.Contents)
        & N"PF.Intransit = Free
        & N"Notify.IOInterruptPending
        & E"I: IOintBlok (I<:N"Notify.IOInterrupts & I.ID = ID)
        & (Direction = OUT => N"Slot
                 <>      N"PF).Status = ReadWrite
```

```
transform SwapOut(Pframe: PageFrameName,
                  ID: RequestName,
                  R: KProcBlok)

refcond R<:ProcessList & CheckFrame(R,Pframe)
        & PF.Status<:S"(ReadOnly,ReadWrite)

effect E"PS:PageSlot (PS.Status = Free & EnqueueSwap(PF, OUT, PS,
                (RecNum(PS.Addr) = 1 => (SearchHAEq, PS.Addr, ()
                        <> (SearchKeyEQ, U"PS1: PageSlot
                                (PS1<:ReadSlots
                                 & TrackAddr(PS1.Addr) = TrackAddr(PS.Addr)
                                 & RecNum(PS1.Addr) =
                                        RecNum(PS.Addr)-1).Key,
                                8)),
                R, ID) )
```

```
transform SwapIn(Pframe: PageFrameName,
                 Slot: PageSlotName,
                 IO: RequestName,
                 R: KprocBlok)

refcond R<:ProcessList & CheckFrame(R, Pframe) & CheckSlot(R, Slot)
        & PS.Status = ReadWrite & PF.Status<:S"(EmptyPage,ReadWrite)

effect EnqueueSwap(PF, IN, PS,
                   (SearchKeyEQ, PS.Key, 8),
                   R, IO)
```

2.11: Status Page Function

This section contains the single function StatusPage, which returns status information about a virtual page to the NKCP owning the page.

```
transform  StatusPage(Vpage:  VirtualPageName  /* VMName + Page Number */ ,
                       R: KProcBlok)

refcond  CheckMPage(R, VPage.VMname, VPage.A, VPage.Vaddr) & R<:ProcessList
         & LET"VP = VirtPage(AS,Vpage.Vaddr)
                 & LET"Ad = RealAddress(AS,Vpage.Vaddr)
         & Ad.2 -> LET"PF = U"PF: PageFrame (PF<:RealPages & PF.Addr = Ad.1)

effect    (AD.2 => N"VP.Change = (VP.Change | PF.Change)
                       & N"VP.Refer = (VP.Refer | PF.Refer)
                     & N"Status.ReadWrite = (PF.Status = ReadWrite)
                     & N"Status.Present = true
          <> ~AD.2 => ~N"Status.Present & N"Status.ReadWrite =true
                 & N"Status.Slot = VP.VSlot)
         & N"Status.Refer = N"VP.Refer & N"Status.Change = N"VP.Change
```

2.12: Storage Key Functions

This section contains the kernel functions that read and write storage keys in virtual pages.

```
transform SetStorageKey(VM: VirtualMachineName,
                        Page:  VPage,
                        Key: KeyInStorage,
                        R: KProcBlok)

refcond  R.Proctype = NKCP & R<:ProcessList & CheckMPage(R, VM, 0, Page)
         & LET"VP = VirtPage(AS,Page)
         & LET"Ad = RealAddress(AS, Page) & VP.Status = ReadWrite

effect  N"VP.SavedKey = Key
 & (Ad.2 => LET"PF = U"PF: PageFrame (PF<:RealPages  & PageFrame.Addr = Ad.1)
                                   & N"PF.RealKey.NoFetch = Key.NoFetch
                                  & N"PF.RealKey.Key = Key.Key
                                  & ~N"PF.RealKey.Change
                                  & ~N"PF.RealKey.Refer
                                   & N"VP.Change =
                                     (VP.Change | PF.RealKey.Change)
                                  & N"VP.Refer = (VP.Refer | PF.RealKey.Refer)
    <> ~ Ad.2 => N"VP.Change = VP.Change & N"VP.Refer = VP.Refer)
```

```
transform ReadStorageKey(VM: VirtualMachineName,
                         Page: 0..8191,
                         R: KProcBlok,
                         Key: KeyInStorage /* output */ )

refcond R.Proctype = NKCP & R<:ProcessList & CheckMPage(R, VM, 0, Page)
        & LET"VP = VirtPage(AS,Page) & LET"Ad = RealAddress(AS, Page)

effect  (Ad.2 => LET"PF = U"PF: PageFrame (PF<:RealPages  & PF.Addr = Ad.1)
                  & N"Key.Key = PF.RealKey.Key
                  & N"Key.NoFetch = PF.RealKey.NoFetch
                  & N"Key.Change = (PF.RealKey.Change | VPChange)
                  & N"Key.Refer = (PF.RealKey.Refer | VP.Refer)
        <> ~RealAddress(A,Page).Valid => N"Key = VirtPage(A,Page).SavedKey)
```

2.13: Lock/Unlock Page Functions

This section contains the kernel functions that manipulate user locks on page frames.

```
transform LockPage(Pframe: PageFrameName,
            R: KProcBlok)

refcond CheckFrame(R, Pframe) & PF.ILOCK = 0 & ~PF.ULOCK

effect N"PF.ULOCK = true
```

```
transform UnlockPage(Pframe: PageFrameName,
                     R: KProcBlok)

refcond   CheckFrame(R, Pframe) & PF.ULOCK

effect   N"PF.ULOCK = false
```

2.14: Scheduling Functions

This section contains the kernel functions which oversee the basic process scheduling provided by the kernel. ScheduleProcess is invoked only by the (semi-trusted) scheduler. The function schedules NKCPs and trusted processes. Once an NKCP is active, it may request the remaining scheduling operations. ReceiveInterrupts and DispatchVM are fairly clear; the former asks the kernel to provide the NKCP with an outstanding interrupt, if one exists. ReleaseCPU signals the termination of activity of the process, and allows the kernel to activate the next available process. ResumeProcess allows an NKCP to modify its activity status bits without relinquishing the CPU to another process. For example, if the NKCP decides to turn on monitoring, it may do so by performing the ResumeProcess kernel call with the appropriate parameters.

```
type InterruptMask = structure of
         (IntvlTimer = boolean,
          Message = boolean,
          ClockComp = boolean,
          CPUtimer = boolean,
          IO = boolean,
          Quantend = boolean)
```

```
transform ResumeProcess(NewPSW: ProgramStatusWord,
                        Gregs: RegisterSet,
                        MON: MonitorClasses,
                        R: KProcBlok)

refcond R<:ProcessList

effect  N"SYSTEM-Gregs = Gregs
        & N"SYSTEM-PSW = NewPSW
        & N"SYSTEM-Cregs =
                ( (K2, K64), R.AddressSpace.Addresses,
                  -1, 0, 0, 0,
                  (false /*l*/, false, false, true,
                    false, false, true, R.Cregs, true),
                  0, MON,
                  (false, false, false, false, 0),   0, 0, 0, 0,
                  (true, true, false, false,
                    false, true, false, false, false),   512)
        & N"VMA = (R.AddressSpace.Addresses, R.Cregs, true, NewPSW,
                    WorkArea, 0, 0, 0, 0)
```

```
transform RECEIVEInterrupts(M: InterruptMask,
                            R: KProcBlok)

refcond R<:ProcessList

effect (R.IntvlTimerInterruptPending & M.IntvlTimer
             => SimulateExternalInterrupt(R, X'80')
       <> R.MessagePending & M.Message & ~R.MessagePendingReceived
             => SimulateExternalInterrupt(R, X'1202')
       <> R.ClockCompInterruptPending & M.ClockComp
             => SimulateExternalInterrupt(R, X'1004')
       <> R.CPUTimerInterruptPending & M.CPUtimer
             => SimulateExternalInterrupt(R, X'1005')
       <> R.IOInterrupts ~= empty & M.IO
             => E1"B:IOintBlok (N"R.IOInterrupts = R.IOInterrupts ~~ S"(B)
                                & SimulateIOInterrupt(R, B.ReqType, B.Id) )
       <> N"R = R)
```

```
transform ReleaseCPU(R: KProcBlok)

refcond R<:ProcessList

effect N"R.Gregs = SYSTEM←Gregs
         & N"R.MON = SYSTEM←Cregs.8
         & N"R.IntvlTimer = SYSTEM←TIMER
         & N"R.CPUTimer = SYSTEM←CPUTimer
                 + (R.TrackingQuantum => R.CPUTimer - R.RemainingQuantum
                                         <> 0)
         & El"P: KProcBlok (P<:ProcessList & N"Requester = P.ProcName
                         & ResumeProcess(P, P.ResumePsw, P.Gregs, P.MON)
                         & N"SYSTEM←TIMER = P.IntvlTimer
                         & N"SYSTEM←CPUTimer =
                                 (P.TrackingQuantum => P.RemainingQuantum
                                                       <> P.CPUTimer)  )
         & PurifySharedSystems
```

```
transform ScheduleProcess(P, R: KProcBlok)

refcond P<:ProcessList & R<:ProcessList & R.ProcName = SCHEDULER

effect N"Requester = P.ProcName & ResumeProcess(P, P.ResumePsw, P.Gregs, P.MON)
         & N"SYSTEM-TIMER = P.IntvlTimer
         & N"SYSTEM-CPUTimer = (P.TrackingQuantum => P.RemainingQuantum
                                                  <> P.CPUTimer)
```

```
transform DispatchVM(VMname: VirtualMachineName,
                  A: AddressSpaceDesignator,
                  NewPsw: ProgramStatusWord,
                  Gregs: RegisterSet,
                  Fregs: FregSet,
                  DAT: DATSpec,
                  MON: HalfMask,
                  PER: PerSpec,
                  VMA: VMASpec, /* AssistE,VProbSt,ISKd,
                                    370d,SVCd,ShadowE,[CPAsstD,]
                                    TimerE,Cregs,VIP*/
                  Time: integer,
                  R: KProcBlok)

refcond R<:ProcessList & R.ProcType = NKCP
        & Time > 0 & Time < MaxTimeSlice*300*256

effect N"SYSTEM←Gregs = Gregs & N"SYSTEM←Fregs = Fregs
        & N'SYSTEM←TIMER = Time
        & N"SYSTEM←PSW = (NewPsw.ProtectionKey,
                          NewPsw.CC, NewPsw.ILC,
                          NewPsw.ProgMask,
                          NewPsw.InstAddr)
        /* R, T, IO, EXT, EC, M, P = 1, W = 0 */
        & N"SYSTEM←Cregs = ( DAT.1, DAT.2, /* Page & Segment Size */
                          AddressSpace(K,A).Addresses, /* CR 1 */,
                          -1, 0, 0, 0,
                  /* VMA Control = VMA Mask >< MICBlokAddress
                          (Furnished by Kernel) */
                          (VMA.1, VMA.2, VMA.3, VMA.4,
                             VMA.5, VMA.6, false, VMA.7),
                          MON, 0,
                  /* PER control = EventMask, REgisterMask, PerStart, PerEnd */
                          (PER.1, PER.2), PER.3, PER.4,
                          0, 0,
                          (true, true, false, false,
                              false, true, false, false, false, false),
                          512)
                  /* RealSegmentTbl, VirtCregs, VIP, VirtPSW,
                           WorkArea, VTimer, 0, 0, 0 */
        & N"KERNEL←MICBLOK = (AS.Addresses,VMA.8.real, VMA.9, NewPsw, Workarea,
                          (~VMA.7 => 0
                           <> Page0Present(AS) => RealAddress(AS,0)+80
                           <> RealAddress(R.AddressSpace, VMblock).VMTIMER),
                          0, 0, 0)
        & CheckASpace(R, VMname, A)
```

2.15: Allocation/Deallocation Functions

This section contains the kernel functions dealing with allocation and deallocation of page frames and page slots. In KVM a page frame is an area of real memory, and a page slot is an area on a direct-access storage device. A page is the contents of a page frame or a page slot,

```
transform FreePageFrame(PF: PageFrame)

refcond PF.Status = ReadWrite &~PF.ULOCK & ~PF.TLOCK
        & PF.ILOCK = 0 & PF.OLOCK = 0 & PF.Intransit = Free

effect  N"PF.Status = FREE & N"PF.VirtList = empty
        & ~N"PF.VirtAddr.Valid
        & A"VPN: VirtualPageName( VPN<:PF.VirtAddr.VaddrList
                -> CheckASpace(U"P: KProcBlok (P.ProcName = PF.Owner),
                                VPN.VMname,
                                VPN.A)
                         & N"AS.PurgeTLBNeeded)
```

```
transform FreeSlot(Slot: PageSlot)

refcond Slot.Status ~<:S"(Free, EmptyPage)

effect N"Slot.Status = Free
```

```
transform ReleaseSlot(Slot: PageSlot,
                      R: KProcBlok)

refcond R<:ProcessList & R.ProcName<:S"(SCHEDULER, INITIATOR)

effect FreeSlot(Slot)
```

```
constant InsertSwap(L: list of SwapBlok,
                    AFTER: SwapBlok,
                    NEW: SwapBlok) ==
          (L..1 = AFTER => L :. NEW
                    <> InsertSwap(L ::.2, AFTER, NEW):. L..1)
```

```
transform ChainPageIO(SR: SwapReq,
                      New: SwapBlok,
                      Where: SwapBlok,
                      R: KProcBlok)

refcond R<:ProcessList & R.ProcName = SCHEDULER
        & SR<:SwapQueue & Where<:SR & ~SR.Active
        & New.Slot.1 = Where.Slot.1

effect N"SR = InsertSwap(SR, Where, New)
```

```
transform ReleasePage(Page: VirtualPageName,
                      R: KProcBlok)

refcond R<:ProcessList & CheckASpace(R, Page.VMname, Page.A)
        & LET"VP = VirtPage(AS, Page.Vaddr) & VP.Status ~= EmptyPage

effect  N"VP.Status = EmptyPage
        & (VP.Valid => FreePageFrame(U"PF: PageFrame (PF.Addr = VP.Raddr) )
                 <> FreeSlot(U"PS: PageSlot( PS.Sname = VP.VSlot
                                             & PS.Owner = R.ProcName) ) )
```

```
transform SetSlot(PF: PageFrame,
                  Slot: PageSlot)

refcond PF.Status<:S"(ReadWrite, ReadOnly) & PF.Intransit ~= IN
        & PF.ILOCK = 0 & PF.OLOCK = 0 & ~PF.ULOCK
        & PS.Status = Free

effect  N"Slot.Status = EmptyPage
        & N"Slot.Owner = PF.Owner & N"Slot.Sname = PF.VirtAddr.VaddrList.1
        & N"PF.VirtAddr.VSlot = N"Slot.Sname
```

```
transform AssignSlot(PF: PageFrame,
                     Slot: PageSlot,
                     R: KProcBlok)

refcond R<:ProcessList & R.ProcName = SCHEDULER

effect SetSlot(PF, Slot)
```

```
transform StealPageFrame(PF: PageFrame,
                         ID: RequestName,
                         Notify,
                         R: KProcBlok)

refcond R<:ProcessList & R.ProcName = SCHEDULER
        & PF.Status = ReadWrite & ~PF.ULOCK & ~PF.TLOCK
        & PF.ILOCK = 0 & PF.OLOCK = 0
        & PF.Intransit ~= IN

effect  FreePageFrame(PF)
        & (PF.VirtAddr.Change ->
                E"PS: PageSlot(PS.Status = Free
                               & EnqueueSwap(PF, OUT, PS, Notify, ID) ) )
```

```
transform GetPageFrame(ID: RequestName,
                       R: KProcBlok,
                       Pframe: PageFrameName)

refcond R<:ProcessList

effect  N"Pframe = R.GetPageCount & N"R.GetPageCount = R.GetPageCount + 1
        & (E"PF: PageFrame (PF.Status = Free) =>
                E1"PF: PageFrame (N"PF.Owner = R.ProcName
                        & N"PF.VirtualName = N"Pframe
                        & N"PF.Status = EmptyPage)
                & N"R.IOInterrupts = R.IOInterrupts || S"(ID)
                & N"R.IOInterruptPending
           <> E"PF: PageFrame (PF.Status = ReadWrite & ~PF.ULOCK & ~PF.TLOCK
                        & PF.ILOCK = 0 & PF.OLOCK = 0)
                => StealPageFrame(PF, ID, R,
                        U"P: KProcBlok (P.ProcName = SCHEDULER
                                & P<:ProcessList) ) )
        & (C"S"PF: PageFrame (PF.Status = Free) < MinFreePages
                        & E"PF: PageFrame (Stealable(PF) )
                => E"PF:PageFrame(Stealable(PF) & StealPageFrame(PF,0,0,
                             U"P: KProcBlok (P.ProcName = SCHEDULER
                                  & P<:ProcessList))))
```

2.16: Spooling Functions

This section contains the kernel functions that support spooling using virtual spool cylinder addresses. All requests for spool-type IO are mapped from virtual to real addresses and, if necessary, delayed until the required real object becomes available.

```
type SlotNumber < integer,
        SpoolCylAddress = RealVolumeId >< CylinderNumber,
        SpoolSlotAddress = SpoolCylAddress >< SlotNumber,
        RealSpoolCyl = structure of
                (Exists = boolean,
                 Addr = SpoolCylAddress,
                 Status = (Free, EmptyPage, InUse))

variable RealCyl(SpoolCylAddress): RealSpoolCyl,
         RealSpoolCylinders:set of RealSpoolCyl
```

```
transform RequestSpoolIO(Direction: (IN,OUT),
                              Slot: SpoolSlotAddress,
                              ID: RequestName,
                              Page: VPage,
                              R: KProcBlok)

refcond R<:ProcessList & R.ProcType = NKCP
        & CheckVPage(R.AddressSpace, Page)
        & LET"RA = RealAddress(R.AddressSpace, Page)
        & RA.2 & LET"PF = U"PF:PageFrame (PF.Addr = RA.1 & PF<:RealPages)
                & PF.Intransit = Free
        & (Direction = IN => PF.OLOCK = 0
                                & PF.Status<:S"(ReadWrite, EmptyPage)
                                & RealCyl(R, Slot.1).Exists
                <> PF.ILOCK = 0 & PF.Status = ReadWrite)

effect (RealCyl(R, Slot.1).Exists => N"RealCyl(R,Slot.1) = RealCyl(R,Slot.1)
                <> E"C: RealSpoolCylinder (Status(C) = Free
                                              & C<:RealSpoolCylinders
                                & N"Status(C) = EmptyPage
                                & N"RealCyl(R, Slot.1).Addr = C.Addr)
                         & N"RealCyl(R, Slot.1).Exists)
        & E"SR: SwapReq, SB: SwapBlok (SR<:N"SwapQueue & SB<:SR
                                & SB.Frame = PF.Addr & SB.ReqType = Direction
                                & SB.Notify = R.ProcName & SB.ID = ID
                                & SB.Slot = (N"RealCyl(R, Slot.1), Slot.2)
                                & SW.Check = (RealSpoolCyl(R, Slot.1).Exists
                                         => NoChecks
                                         <> Clear) )
```

```
transform ReleaseSpoolCylinder(V: SpoolCylinderAddress,
                               R: KProcBlok)

refcond RealCyl(R, V).Exists & R.Proctype = NKCP & R<:ProcessList

effect ~N"RealSpoolCyl(R, V).Exists
       & N"SpoolStatus(RealSpoolCyl(R,V).Addr) = Free
```

2.17: Send/Receive Message Functions

This section contains the kernel functions that provide communication between processes. Trusted processes can both send and receive messages from all other processes. Semi- trusted processes can only receive messages. NKCPs can send messages to both semi-trusted and trusted processes, but can receive only from trusted processes.

```
type MessageId = structure of
          (MsgType = (OneWay  TwoWay, Reply),
           /* following meaningful only if MsgType ~= OneWay */
           Name = requestName,
           /* following meaningful only if MsgType = TwoWay */
           Time = UnsignedDoubleWord)

type MsgBlok = structure of
          (Sender = ProcessName,
           MessageId = MessageId,
           Contents = T"L: list of Byte (C"L <= PageSize*1024),
           MsgLength = T"I: integer (I >= 0 & I <= PageSize*1024)  )
```

```
transform ReceiveMessage(Buffer: Addres370,
                         R: KProcBlok,
                         MessageLength: ByteOffset,
                         Sender: ProcessName,
                         Id: MessageId)
refcond R.MessagePendingReceived & R.MessageQueue ~= nil
        & CheckVPage(R.AddressSpace, Buffer.1)
        & Buffer.2 + MessageLength <= PageSize * 1024
        & LET"Ad = RealAddres(R.AddressSpace, Buffer.1)
        &Ad.2
        &LET"PF = U"PF: PageFrame (PF<:RealPages & PF.Addr = Ad.1)
        & PF.Status = ReadWrite

effect ~N"R.MessagePendingReceived & N"R.MessageQueue = R.MessageQueue:2
        & N"Id = R.MessageQueue.1.MessageId
        & N"Sender = R.MessageQueue.1.Sender
        & N"MessageLength = R.MessageQueue.1.Length
        & A"i:ByteOffset (i < N"MessageLength ->
                        N"PF.Contents.(i+Buffer.2) =
                               R.MessageQueue.1.Contents.i)
        & N"R.MessagePending = (N"R.MessageQueue ~= nil)
```

```
transform SendMessage(ID: MessageId,
                 MessageLength: ByteOffset,
                 Message: Address370,
                 D,
                 R: KProcBlok)

rafcond R<:ProcessList & D<:ProcessList
         & (SendMessageAllowed(R.ProcName, D.ProcName) | R.Proctype = TRUSTED)
         & CheckVPage(R.AddressSpace, Message.1)
         & Message.2 + MessageLength <= PageSize * 1024
         & LET"Ad = RealAddress(R.AddressSpace, Message.1)
         & Ad.2
         & LET"PF = U"PF: PageFrame (PF<:RealPages & PF.Addr = Ad.1)

effect (R.Id.MsgType = Reply & ~E"B:ClockBlok
                         (B<:ClockCompQueue & B.Proc = D.ProcName
                          & B.Reason = MessageReq & B.Id = ID.Name)
                 => N"D = D & N"ClockCompQueue = ClockCompQueue & N"R = R
         <> N"D.MessagePending
           & E1"M:MsgBlok (N"D.MessageQueue = D.MessageQueue || S"(M)
                                 & M.Sender = R.ProcName
                         & M.Length = MessageLength & M.MessageId = ID
                         & A"i:ByteOffset (i < MessageLength ->
                                 M.Contents.i = PF.Contents.(Message.2+i) ))
                         & M.MessageId.Name = (   ID.MsgType = OneWay => 0
                                                         <> ID.Name)
                         & N"D.MessageQueue = D.MessageQueue :. M)
           & (ID.MsgType = OneWay => N"ClockCompQueue = ClockCompQueue
              <> ID.MsgType = TwoWay => E"B:ClockBlok
                                   (N"ClockCompQueue = ClockCompQueue || S"(B)
                                  & B.Proc = R.ProcName
                                  & B.Reason = MessageReq & B.ID = ID.Name
                                  & B.Time = SYSTEM←TODClock + ID.Time)
              <> ID.MsgType = Reply => E"B: ClockBlock
                         (B<:ClockCompQueue & B.Proc = R.ProcName
                                 & B.Reason = MessageReq & B.ID = ID.Name
                                  & N"ClockCompQueue =
                                        ClockCompQueue ~~ S"(B) )   )
```

```
transform ReadCPUId(ID: ProcessorId,
                    R: KProcBlok)

effect N"ID = SYSTEM←CPUId
```

2.18: Timing Functions

This section contains the kernel functions that control and support three of the four timers provided by the S/370. Each NKCP is provided with its own set of virtual timers, which are mapped onto the real set when the NKCP is active. The three provided timers are the interval timer, the CPU timer, and the clock comparator. The latter is used for two purposes in KVM: the kernel uses it to time out two-way messages, and the NKCP may also use it for its own purposes. The kernel maintains a queue of requests to set the clock comparator. The fourth timer, the time-of-day clock, is read by a non-privileged instruction. KVM/370 provides no facility to write it during normal operation, but treats it as a read-only object.

```
constant Delta = 1000 /* Time in microseconds to handle a
                         Kernel Call and redispatch */
variable SETTime: SignedDoubleword

type ClockBlok = structure of
        (Time = UnSignedDoubleWord,
         Proc = ProcessName,
         Reason = (ClockReq, MessageReq),
         /* following meaningful only if Reason = MessageReq */
         Id = RequestName)

variable ClockCompQueue: set of ClockBlok,
        SystemClockCompId: ClockBlok
```

```
transform SetClockComparator(TIME: UnsignedDoubleword,
                            R: KProcBlok)

effect E"B: ClockBlok (B<:N"ClockCompQueue & B.Proc = Requester
                  & B.Time = TIME & B.Reason = ClockReq
                  & A"z: ClockBlock (z<:N"ClockCompQueue
                                  & z.Proc = B.Proc & z.Reason = ClockReq
                                          -> z = B))
    & N"R.ClockCompInterruptPending = False
    & E1"x<:N"ClockCompQueue (N"SYSTEMClockComparator = x.Time
                                  & N"SystemClockCompId = x)
    & A"y<:N"ClockCompQueue  (y.Time >= N"SYSTEMClockComparator)
```

```
transform ClockComparatorInterrupt()

refcond SystemClockCompId<:ClockCompQueue
       & E1"P: KProcBlok (P<:ProcessList & P.Name = SystemClockCompId.Proc)
        & LET"B = U"ClockBlok (SystemClockCompId = B)

   /* Note: ClockComparatorInterrupt occurs when
        SYSTEM←ClockComparator < SYSTEM←TODClock */

effect B.Reason = ClockReq => N"P.ClockCompInterruptPending
             & N"ClockCompQueue = ClockCompQueue ~~ S"(B)
        <> N"P.MessagePending = P.MessagePending ; B.Id )
      & (   N"ClockCompQueue = nil =>
                 N"SystemClockComparator = MaxClockCompValue
        <> N"ClockCompQueue ~= nil => E"x<:N"ClockCompQueue
                 (A"y<:N"ClockCompQueue (y.Time >= x.Time)
                             & N"SystemClockComparator = x.Time
                                 & N"SystemClockCompId = X))



CLOCK COMPARATOR INVARIANTS: A"x:ClockBlok (x<:ClockCompQueue ->
                                        x.Time >= SYSTEMClockComparator)
        A"y,z:ClockBlok (y<:ClockCompQueue
                         & z<:ClockCompQueue
                         & z.Proc = y.Proc
                         & z.Reason = ClockReq
                         & y.Reason= ClockReq
                                -> z = y)
ClockCompQueue ~= nil -> SystemClockCompId<:ClockCompQueue
                        & SystemClockCompId.Time = SYSTEMClockComparator
```

```
transform SetCPUTimer(Time: SignedDoubleword,
                      R: KProcBlok)

refcond R.Proctype = NKCP

effect N"R.CPUTimer = Time
    & (Time < N"R.RemainingQuantum => N"R.TrackingQuantum = False
                                        & N"SETTime = Time
      <> Time >= N"R.RemainingQuantum => N"R.TrackingQuantum = True
                                        & N"SETTime = N"R.RemainingQuantum)
    & N"SYSTEM←CPUTimer <= N"SETTime
    & N"SYSTEM←CPUTimer >= N"SETTime - (Delta * 4096)
    & N"R.CPUTimerInterruptPending = False
    & (R.TrackingQuantum => N"R.RemainingQuantum = SYSTEM←CPUTimer
      <>~R.TrackingQuantum => N"R.RemainingQuantum =
                              R.RemainingQuantum-R.CPUTimer
```

```
transform ReadCPUTimer(R: KProcBlok,
                Time: SignedDoubleword /* output */ )

refcond  R.Proctype = NKCP

effect  (R.TrackingQuantum => N"Time = R.CPUTimer -
                                         R.RemainingQuantum +
                                         SYSTEM←CPUTimer
       <>~R.TrackingQuantum => N"Time = SYSTEM←CPUTimer)

CPU TIMER INVARIANTS:  R.Proctype = NKCP -> (SYSTEM←CPUTimer <=
                                                  R.RemainingQuantum
                                              & SYSTEM-CPUTimer <= R.CPUTimer
                                              & R.TrackingQuantum <->
                                      R.RemainingQuantum <= R.CPUTimer)
```

```
transform CPUTimerInterrupt()

refcond R.Proctype = NKCP

     /* CPU timer interrupt occurs when R.Proctype = NKCP
                                                & SYSTEM←CPUTimer < 0 */

effect (R.TrackingQuantum => N"R.CPUTimer = R.CPUTimer -
                                             R.RemainingQuantum +
                                             SYSTEM←CPUTimer
                 & N"R.RemainingQuantum = MaxCPUTimerValue
                 & N"SETTime = N"R.CPUTimer
                 & N"R.TrackingQuantum = False & N"R.QuantumEnded = True
       <>~R.TrackingQuantum => N"R.RemainingQuantum = R.RemainingQuantum
                                            - R.CPUTimer + SYSTEM←CPUTimer
               & N"R.CPUTimer = MaxCPUTimerValue
               & N"SETTime = N"R.RemainingQuantum
               & N"R.TrackingQuantum = True
                & N"R.CPUTimerInterruptPending = True)
       & N"SYSTEM←CPUTimer <= N"SETTime
                                   & N"SYSTEM←CPUTimer >=
                                     N"SETTime - (Delta * 4096)
```

```
transform SetIntervalTimer(Time: Integer,
                           R: KProcBlok)

refcond  R.Proctype = NKCP

effect
   N"SYSTEM←TIMER <= Time
   & N"SYSTEM←TIMER >= Time - (Delta * 3 / 40)
```

```
transform ReadIntervalTimer(R: KProcBlok,
                            Time: integer /* output */ )

refcond  R.Proctype = NKCP

effect  N"Time <= SYSTEM←TIMER
       & N"Time <= SYSTEM←TIMER + (Delta * 3 / 40)
```

```
transform IntervalTimerInterrupt()

refcond  R.Proctype = NKCP & SYSTEM←TIMER < 0

   /* An interval timer interrupt occurs when the machine timer update:
        N"SYSTEM←TIMER = SYSTEM←TIMER - INTVL
     causes
        N"SYSTEM←TIMER < 0 & SYSTEM←TIMER >= 0 */

effect  N"R.IntvlTimerInterruptPending = True
```

2.19: IO Functions

This section contains the kernel functions that perform real IO. Details of the functions' operation may be found in the preliminary design document of KVM/370, TM-5855/200/00.

```
type CCWAccessFlags = structure of
(Access = (Write, Read),
         TIC = boolean, DataTransfer = boolean, DataChaining = boolean,
         MultiTrack = boolean, Seek = boolean)

type IOintBlok = structure of
        (ID: RequestName,
         CSW: ChannelStatusWord)

type LockedBox, PreprocessedChannelProgram, SeekArg, IDAW,
        CCWCount = T"I: integer (0 <= I & I < 65536)

variable IORequestCount: integer, LockedBoxList: set of LockedBox
constant LookCCW(PreprocessedChannelProgram, CCWCount): ChannelCommandWord,
        CheckTIC(PreprocessedChannelProgram,CCWCount,ByteOffset): boolean,
        LookIDA(PreprocessedChannelProgram,Address370,VPage): IDAW,
        IDACount(PreprocessedChannelProgram, CCWCount): VPage,
        LookSeek(PreprocessedChannelProgram, Address370):SeekArg
variable IOFLAGS(DeviceAddress, CCWOpCode):CCWAccessFlags
```

```
transform RequestIO(Device: DeviceAddress   /* 0..4095 */ ,
                    ChannelProgram: PreprocessedChannelProgram
                            /* includes CCWs, IDAW list,
                               search and seek arguments */
                    VM: VirtualMachineName
                    ID: RequestName,
                    R: KProcBlok)

refcond  LET"G = GrantedAccess (Device, Requester) & CheckASpace(R,VM,0)
         & A"i:integer (i > 0 & i <=  ChannelProgram.NCCWs ->
                LET"CCW = LookCCW(ChannelProgram,i)
               & (i < ChannelProgram.NCCWs -> CCW.ChainCommand)
               & LET"F = IOFlags(Device, CCW.Opcode)
                & (F.Access = Write -> G.Access = ReadWrite)
               & (F.TIC -> CCW.Addr < ChannelProgram.NCCWs
                          & i > 1 & Modif(CCW) = nil
                          & ~LookCCW(ChannelProgram,i-1).DataChain
                          & CCW.Count = 1
                          & CheckTIC(ChannelProgram,CCW.Addr,i))
               & (F.DataTransfer =>
                          A"ii:integer (ii<:CCW.DataChainExtensions
                                           & ii > 0 ->
                          LET"CCWX = LookCCW(ChannelProgram,ii)
                                              & CCWX.Opcode = CCW.Opcode
                                               & CCWX.IndirectAccess)
                                      & A"IDA<:CCW.IDAList
                                              (Let"Ad = RealAddress(A,IDA.1)
                                                  & Ad.2
                                              & LET"PF = U"PF:PageFrame
                                                (PF.Addr = Ad.1
                                                  & PF<:RealPages)
                                              & (   F.Access = Write =>
                                                 PF.ILOCK = 0
                                                    & A"j:integer (j > 0
                                          & j <= ChannelProgram.NCCWs ->
         (i = j | LET"CCW2 = LookCCW(ChannelProgram,j)
               & (CCW2.Skip | IOFlags(Device,CCW2.Opcode).Access ~= Read
                          | A"IDA2<:CCW2.IDAList (RealAddress(A,IDA2.1) ~=
                                                   Ad.1) )  ))
                                                 <> F.Access = Read =>
                                                         CCW.Skip |
                                                               PF.OLOCK = 0
                                                               & A"k:integer
         (k > 0 & k <= ChannelProgram.NCCWs ->
         (i = k | LET"CCW3 = LookCCW(ChannelProgram,k)
         & IOFlags(Device,CCW3.Opcode).Access = Read ->
                   A"IDA3<:CCW3.IDAWords(RealAddress(A,IDA3.1).1 ~= Ad.1)) )
                     <> ~F.DataTransfer => ~CCW.IndirectDataAccess)
                 &(CCW.ChainData -> F.DataChaining)
                 & F.MultiTrack -> ~G.Horizontal
                 &(F.Control -> (CCW.Count = 1 & CCW.SLI))
                 &(G.Class = DASD =>
                      (F.Seek ->
                       (G.FullDisk => True
                   <>G.Horizontal => CCW.SeekAddr.Head<:G.Tracks
                   <>G.Vertical => CCW.SeekAddress.Cyl<:G.Cyls))
                 &(i=1 -> (CCW.Opcode = Seek | CCW.Opcode = SeekCylinder))
                 &(i=2 -> (CCW.Opcode = SetFileMask
                          & LET FM = CCW.FileMask
                          &(G.Access = ReadOnly =>
                                   FM.PermitWrite = InhibitWrite
                           <>G.Access~= ReadOnly =>
                                   FM.PermitWrite ~= PermitAllWrite)
                          &(G.Horizontal => FM.PermitSeek =
                                   InhibitAllSeek
                           <>G.Vertical => FM.PermitSeek ~= PermitAllSeek
                                         & FM.PermitSeek ~= PermitCyl
                           <>G.FullDisk => FM.PermitSeek ~= PermitAllSeek
                           <>G.Dedicate => true)
                          & FM.InhibitDiagWrite = true))
             <> True))))
```

```
effect   N"IORequestCount = IORequestCount + 1

       & E"LB: LockedBox (N"LockedBoxList = LockedBoxList || S"(LB)
                   & LB.Num = IORequestCount
                   & LB.IO = IO
                   & LB.NCCWs = ChannelProgram.NCCWs
                 & LB.Dev = Device & LB.Notify = R.ProcName
                 & A"LBX<:LockedBoxList (LBX.Num < LB.Num)
                 & A"i:integer (i > 0 & i <= ChannelProgram.NCCWs
                   -> LET"CCWA = LookCCW(ChannelProgram,i)
                        & LET"CCWN = (LB.ChanProg.i)

                        & LET"FF = IOFlags(Device,CCWA.Opcode)

                        & CCWN.Opcode = CCWA.Opcode & Modif(CCWN) = Modif (CCWA.)
                        & CCWN.Count = CCWA.Count
                        & CCWN.Addr = CCWA.Addr + LOC(LB)

                        & (F.DataTransfer => A"j:integer (j > 0
                                      & j <= C"CCWA.IDAList
                              -> LET"WN = CCWN.IDAList.j
                              & LET"WA = CCWA.IDAList.j
                              & WN.2 = WA.2
                              & LET"AdN = RealAddress(A,WA.1)
                              & WN.1 = AdN.1
                              & LET"PFN = U"PF:PageFrame(PF<:RealPages
                                                & PF.Addr = AdN.1)
                              & (F.Access = Write => N"PFN.OLOCK = PFN.OLOCK + 1
                                 <> F.Access = Read & ~CCWA.Skip =>
                                                N"PFN.ILOCK = PFN.ILOCK + 1
                                                & ~N"PFN.ULOCK & ~N"PFN.TLOCK))
                          <>F.Seek => LookSeek(ChannelProgram,CCWA.Addr) =
                              LookSeek(LB.ChanProg,CCWN.Addr)

                              <>~F.Access & ~F.Seek => True)))
```

```
transform StartIO(LockedBoxId: integer,
                  R: KProcBlok)

refcond R<:ProcessList & R.ProcName = SCHEDULER
         & LET"L == LB: LockedBox(LB<:LockedBoxList & LB.Num = LockedBoxID)
         & ~InProcess(L)
         & A"B: LockedBox (B<:LockedBoxList & InProcess(B)
                                     -> B.Dev ~= L.Dev)

effect N"InProcess(L) & MachineDependent

   /* Eventual IO Daemon effect:
         ~N"InProcess(L) & LET"P == U"B:KProcBlok (B.ProcName = L.Notify)
         & N"P.IOInterruptPending
         & E"B:IOintBlok (B<:N"P.IOinterrupts & B.IO = L.IO & B.Dev = L.Dev)
         & A"C: LBCCW (C<:L.ChanProg
                         -> LET"F = IOFlags(L.Dev, C.OpCode)
                           & (F.DataTransfer -> A"W: Address370
                                  (W<:C.IDAList -> LET"PF = U"P: PageFrame
                                    (P<:RealPages & P.Addr = W.1)
& (F.Access = Write => N"PF.OLOCK = PF.OLOCK - 1
   <> F.Access = Read => N"PF.ILOCK = PF.ILOCK - 1
               <> true))))
         & MachineDependent */
```

```
transform CancelIO(ID: RequestName,
                   R: KProcBlok)

refcond R<:ProcessList
        & E1"LB: LockedBox (LB<:LockedBoxList
                            & LB.Notify = R.ProcName
                            & LB.ID = ID)

effect E"LB: LockedBox (LB<:LockedBoxList
                        & LB.Notify = R.ProcName
                        & LB.ID = ID
                        & N"LockedBoxList = LockedBoxList ~~ S"(LB)
                        & (InProcess(LB) -> MachineDependent)
                        & ~N"InProcess(LB)
                        & A"C: LBCCW (C<:LB. ChanProg ->
                               LET"F = IOFlags(LB.Dev.C.OpCode)
                        & F.DataTransfer -> A"W: Address370
                               (W<:C.IDAList -> LET"PF = U"PF:PageFrame
                               (PF<:RealPages & PF.Addr = W.1)
                & (F.Access = Write => N"PF.OLOCK = PF.OLOCK -1
                  <> F.Access = Read => N"PF.ILOCK = PF.ILOCK -1
                  <> true))))
```

```
transform WaitIO(Device: DeviceAddress,
                 ChannelProgram: PreprocessedChannelProgram,
                 R: KProcBlok)

refcond R<:ProcessList
          & R.ProcType ~= NKCP
          & LET"G == GrantedAccess(Device, R.ProcName)
          & A"C:ChannelControlWord (C<:ChannelProgram.CCWs ->
               LET"F == IOFlags(Device,C.Opcode)
               & ~F.TIC
               & A"W: IDAW (W<:C.IDAlist
                    -> LET"RA = RealAddress(R.AddressSpace, W.1)
                      & RA.2
                      & LET"PF = U"P:PageFrame (P<:RealPages
                                                  & P.Addr = RA.2)
                      & (C.ChainData -> F.DataChaining)
                      & (F.MultiTrack -> ~G.Horizontal)
                      & (F.Access = Write => PF.ILOCK = 0
                             & A"j:integer (j > 0
                                      & j < ChannelProgram.NCCWs
               -> LET"C2 = LookCCW(ChannelProgram,j)
                  & (C2.SKIP | IOFlags(Device,C2.OpCode).Access ~= Read
                           | A"W2:IDAW
          (W2<:C2.IDAList -> RealAddress(R.AddressSpace,W2.1).1 ~= RA.1)))
                            <> F.Access = Read => C.SKIP |
                                      PF.OLOCK = 0
                                      & A"K:integer
           (K > 0 & K < ChannelProgram.NCCWs
               -> LET"C3 = LookCCW(ChannelProgram,K)
                  & (IOFlags(Device,C3.OpCode).Access ~= Read
                     | A"W3:IDAW(W3<:C3.IDAWords ->
                            RealAddress(R.AddressSpace,W3.1).1 ~= RA.1)))
                           <> true)
                      & (F.DASD -> G.FullDisk)))

effect MachineDependent
```

3.1.1: Operator Process Informal Description

This section contains the informal description of the Operator Process of KVM/370.

Overview

The Operator Process is linked to the operator's console at system initialization. The Operator Process thereafter accepts operator commands, and routes the command either to the affected NKCP, or to the trusted process which is to perform the actions dictated by the command. For the most part, the Operator Process merely routes the message, with little or no notice of the details of the command. The few commands requiring further processing are noted below. Necessarily, the Operator Process must know where the command is to be sent for processing. This often involves mapping a command operand such as a user id into an NKCP id. The Operator Process sends requests to the Authorization Process to perform such mappings. The VM operator commands have been separated by function into four categories: Miscellaneous, Spooling and File Control, Device Control, and Program Analysis and Monitor functions. These categories are described below.

KVM/370 Operator Commands
(see notes at end of table)

Command	S M	Resp Exp	Map	Destin- ation(s)	Category
AUTOLOG	S	Y	N	AuthProcess	1,1
DIAL	not	legal			0,0
FORCE←USERID	S	N(1)	U->N	NKCP	☆3,0
FORCE←ALL	M	N(1)	N	all NKCPs	☆2,0
INDICATE←LOAD	M	Y	N	all NKCPs	2,3
INDICATE←QUEUES	M	Y	N	all NKCPs	2,4
INDICATE←IO	S	Y	N	I/O Scheduler	1,1
INDICATE←PAGING	S	Y	N	I/O Paging Scheduler	1,1
INDICATE←USER	S	Y	U->N	NKCP	3,1
LOGON	not	legal			0,0
LOGOFF	not	legal			
HALT	S	(2)	N	Kernel	5,0
LOCK←USERID	S	Y	U->N	NKCP	3,1
LOCK←SYSTEM	S	Y	N	NKCP	1,1
MESSAGE←USERID	S	N	U->N	NKCP	☆3,0
MESSAGE←ALL	M	N	N	all NKCPs	☆2,0
MONITOR←NKCP	S	Y	N	NKCP	1,1
MONITOR←ALL	M	Y	N	all NKCPs	2,2
NETWORK	S	Y	N	Network Process	1,1
QUERY←PAGING	S	Y	N	I/O Paging Scheduler	1,1
QUERY←PRIORITY	S	Y	U->N	NKCP	3,1
QUERY←SASSIST←NKCP	S	Y	N	NKCP	1,1
QUERY←SASSIST←ALL	M	Y	N	all NKCPs	2,2
QUERY←DASD	S	Y	N	AuthProcess	1,1
QUERY←TAPES	S	Y	N	URProcess	1,1
QUERY←LINES	S	Y	N	AuthProcess	1,1
QUERY←GRAF	S	Y	N	AuthProcess	1,1
QUERY←SYSTEM	S	Y	N	AuthProcess	1,1
QUERY←NAMES	S	Y	N	AuthProcess	1,1
QUERY←USERS	S	Y	N	AuthProcess	1,1
QUERY←USERID	S	Y	N	AuthProcess	1,1
QUERY←UR	S	Y	N	URProcess	1,1
QUERY←ALL	M	Y	N	AuthProcess, URProcess, Kernel	9,2

Command	S M	Resp Exp	Map	Destin-ation(s)	Category
QUERY←DUMP	S	Y	N	Dump Processor	1,1
QUERY←TDSK	M	Y	N	all NKCPs	2,2
QUERY←STORAGE	S	(2)	N	Kernel	5,0
QUERY←RADDR	S	Y	R->T	URProcess or AuthProcess	4,1
QUERY←FILES	S	Y	N	URProcess	1,1
QUERY←READER	S	Y	N	URProcess	1,1
QUERY←PRINTER	S	Y	N	URProcess	1,1
QUERY←PUNCH	S	Y	N	URProcess	1,1
QUERY←HOLD	S	Y	N	URProcess	1,1
QUERY←LOGMSG	processed directly				8,0
SET←FAVORED	S	Y	U->N	NKCP	3,1
SET←RESERVED	S	Y	U->N	NKCP	3,1
SET←PRIORITY	S	Y	U->N	NKCP	3,1
SET←SASSIST←NKCP	S	Y	N	NKCP	1,1
SET←SASSIST←ALL	M	Y	N	all NKCPs	2,2
SET←LOGMSG	processed directly				8,0
SET←DUMP	S	N	N	Dump Processor	☆1,0
SET←RECORD	not legal				0,0
SET←MODE	not legal				0,0
SHUTDOWN	M	N	N	AuthProcess, URProcess, all NKCPs	10,0
SLEEP	not legal				0,0
UNLOCK←USERID	S	Y	U->N	NKCP	3,1
UNLOCK←SYSTEM	S	Y	N	NKCP	1,1
UNLOCK←VIRT	not legal				0,0
WARNING←USERID	S	N	U->N	NKCP	☆3,0
WARNING←ALL	M	N	N	NKCP	☆2,0
BACKSPAC	S	Y	N	URProcess	1,1
CHANGE	S	Y	N	URProcess	1,1
DRAIN	S	Y	N	URProcess	1,1
FLUSH	S	Y	N	URProcess	1,1
FREE	S	Y	N	URProcess	1,1
HOLD	S	Y	N	URProcess	1,1
ORDER	S	Y	N	URProcess	1,1
PURGE	S	Y	N	URProcess	1,1
REPEAT	S	Y	N	URProcess	1,1
SPACE	S	Y	N	URProcess	1,1
START	S	Y	N	URProcess	1,1
TRANSFER	S	Y	N	URProcess	1,1

Command	S M	Resp Exp	Map	Destin-ation(s)	Category
ATTACH←CHANNEL	not	legal			0,0
ATTACH←RADDR	S	Y	R->T	URProcess or AuthProcess	4,1
DETACH←CHANNEL	not	legal			0,0
DETACH←RADDR	S	Y	R->T	URProcess or AuthProcess	4,1
DISABLE	S	Y	N	Network Process	1,1
DISCONN	not	legal			0,0
ENABLE	S	Y	N	Network Process	1,1
LOADBUF	S	Y	N	URProcess	1,1
VARY	S	Y	R->T	URProcess or AuthProcess	4,1
ACNT←USERIDS	Both	Y	U->N	some NKCPs	7,2
ACNT←ALL	M	Y	N	all NKCPs	2,2
DCP	S	Y	N	NKCP	1,1
DMCP	S	Y	N	NKCP	1,1
LOCATE←USERID	S	Y	U->N	NKCP	3,1
LOCATE←RADDR	S	Y	R->T	URProcess or AuthProcess	4,1
SAVESYS	not	legal			0,0
STCP	S	Y	N	NKCP	1,1

Notes:

The column headings are as follows:

Command = command name
S/M = single message versus multiple messages sent
Resp Exp = response expected (Y=yes, N=no)
Map = mappings performed, if any:
    U -> N = user id to NKCP id
    R -> T = real device address to device type
Destination = to what processes the messages are sent
    AuthProcess = Authorization Process,
    URProcess = Unit Record Process

Category = ordered pair of (input category, output category). The category refers to the classification of the command in the formal specification. Each operator command is separated into two equivalence classes by

- the processing required when the command is first received, and

- the processing required to compute one response from all the responses to messages sent in the processing of the original command.

(1) FORCE command responses are provided by the Authorization Process as unsolicited messages rather than solicited responses because the number of expected responses is not known and because it simplifies the interaction.

(2) A response is provided immediately -- no message response is needed.

OPERATOR COMMANDS: MISCELLANEOUS FUNCTIONS

AUTOLOG (A,B)

"Automatically log on and load a virtual machine with IPL defined in its directory."

[extend syntax to provide NKCP id (security level)]
reflect message to Authorization Process
print response

DIAL (Any)

"Logically attach a user terminal to a multi-access virtual system."

not allowed as an operator command in KVM/370

FORCE (A)

"Force the logoff of any virtual machine."

```
[extend syntax to provide "ALL" operand]
user id - map user id to NKCP id
          reflect message to NKCP
          no response expected
          [response from Authorization Process]

ALL -     reflect message to each NKCP
          no response expected [separate responses
             from Authorization Process]
```

INDICATE (E)

"Display use of and contention for major system resources."

```
LOAD, QUEUES -
         reflect message to each NKCP
         gather responses
         compute "average" response
         print computed response

I/O -    reflect message to I/O Scheduler
         print I/O Scheduler response
```

```
PAGING - reflect message to I/O Paging Scheduler
         print I/O Paging Scheduler response

USER -   map user id to NKCP id
         reflect message to NKCP
         print NKCP response
```

LOGON (Any)

"Gain access to a virtual system from a user terminal."

not allowed as an operator command in KVM/370

LOGOFF (Any)

"Terminate user activity on a virtual machine."

reflect message to Authorization Process
[response from Authorization Process]

HALT (A)

"Stop any active channel program on a specified device."

Kernel Call: HALT
print Kernel response

LOCK (A)

"Lock a user's pages in processor storage."

```
user id - map user id to NKCP id
          reflect message to NKCP
          print NKCP response

SYSTEM -  [extend syntax to provide NKCP id
             (security level)]
          reflect message to NKCP
          print NKCP response
```

MESSAGE (Any)

"Send a specified message from one virtual machine to another."

```
user id - map user id to NKCP id
          reflect message to NKCP
          no response to operator
```

```
ALL -      reflect message to each NKCP
           no response to operator
```

MONITOR (A,E)
[Also listed under PROGRAM ANALYSIS AND MONITOR FUNCTIONS]

"Initiate or terminate the recording of events that occur in the real machine."

```
[extend syntax to provide NKCP id
   (security level) / "ALL" operand]

NKCP id - reflect message to NKCP
          print NKCP response

ALL -     reflect message to each NKCP
          gather responses
          print NKCP response
```

NETWORK (A,B,F)
[Also listed under DEVICE CONTROL FUNCTIONS]

"Provide loading, dumping, tracing, and other functions for the 3704/3705 Communications Controller."

```
reflect message to Network Process
print Network Process response
```

QUERY

"Provide status information on the real or virtual machine, and miscellaneous CP functions."

```
PAGING -   reflect message to I/O Paging Scheduler
           print I/O Paging Scheduler response

PRIORITY - map user id to NKCP id
           reflect message to NKCP
           print NKCP response

SASSIST -  [extend syntax to provide NKCP id
              (security level) / "ALL" sub-operand]

           NKCP id - reflect message to NKCP
                     print NKCP response

           ALL -     reflect message to each NKCP
                     gather responses
                     print amalgamated response
```

```
DASD, LINES, GRAF,
SYSTEM, NAMES, USERS, user id -
                reflect message to Authorization Process
                print Authorization Process response

UR -            reflect message to URProcess
                print URProcess response

ALL -           reflect message to Authorization Process
                   and URProcess
                Kernel Call: STORAGE
                gather responses
                print amalgamated response

TDSK -          reflect message to each NKCP
                gather responses
                print amalgamated response

STORAGE -       Kernel Call: STORAGE
                print Kernel response

raddr -         map raddr to device type
                if device is unit record device:
                   reflect message to URProcess
                   print URProcess response
                else:
                   reflect message to Authorization Process
                   print Authorization Process response

TAPES, FILES, READER, PRINTER, PUNCH, HOLD -
                [extend syntax to provide NKCP id
                   (security level) / "ALL" operand]

                reflect message to URProcess
                print URProcess response

LOGMSG -        print log message

DUMP -          reflect message to Dump Processor
                print Dump Processor response
```

SET (A,B,F)

"Establish system parameters for virtual and real machines, as well as other VM/370 values."

```
FAVORED, RESERVE, PRIORITY -
              map user id to NKCP id
              reflect message to NKCP
              print NKCP response
```

```
SASSIST - [extend syntax to provide NKCP id
              (security level) / "ALL" sub-operand]

                NKCP id - reflect message to NKCP
                          print NKCP response

                ALL -     reflect message to each NKCP
                          gather responses
                          print amalgamated response

LOGMSG -        NULL - set log message to empty string

                nn -   read next console line, and
                          make it line nn in the current
                          log message

DUMP -          reflect message to Dump Processor
                [no response to operator]

RECORD, MODE -
                not allowed in KVM/370
```

SHUTDOWN (A)

"Terminate VM/370 activity in an orderly manner and checkpoint the system."

Reflect message to Authorization Process, URProcess, and each NKCP
no response to operator

SLEEP (Any)

"Place the virtual machine in a dormant state, with the keyboard locked."

Not allowed as an operator command in KVM/370

UNLOCK (A)

"Release previously locked page frames of real storage."

```
user id - map user id to NKCP id
          reflect message to NKCP
          print NKCP response

SYSTEM, VIRT -
          not allowed in KVM/370
```

WARNING (A,B)

"Transmit high priority messages or warnings to a user or all users."

```
user id - map user id to NKCP id
          reflect message to NKCP
          no response to operator

ALL -     reflect message to each NKCP
          no response to operator
```

O P E R A T O R   C O M M A N D S: SPOOLING AND FILE CONTROL FUNCTIONS

All Spooling and File Control Commands are reflected to the Unit Record Process which makes the necessary response. See the Unit Record Process Description for processing details.

BACKSPAC (D)
CHANGE (D)
DRAIN (D)
FLUSH (D)
FREE (D)
HOLD (D)
ORDER (D)
PURGE (D)
REPEAT (D)
SPACE (D)
START (D)
TRANSFER (D)

O P E R A T O R C O M M A N D S: DEVICE CONTROL FUNCTIONS

ATTACH (B)

"Attach a real device to a virtual machine or the real system."

```
CHANNEL - not allowed in KVM/370

raddr -   map raddr to device type
          if device type is unit record:
             reflect message to URProcess
             print URProcess response
          else:
             reflect message to Authorization Process
             print Authorization Process response
```

DETACH (B)

"Remove a real or virtual device or channel from a virtual machine or the real system."

```
CHANNEL - not allowed in KVM/370

raddr -   map raddr to device type
          if device type is unit record:
             reflect message to URProcess
             print URProcess response
          else:
             reflect message to Authorization Process
             print Authorization Process response
```

DISABLE (A,B)

"Disable direct or switched communication lines from the VM/370 system."

```
reflect message to Network Process
print Network Process response
```

DISCONN (Any)

"Disconnect the terminal from the user's virtual machine."

not allowed as an operator command in KVM/370

ENABLE (A,B)

"Connect specified communication lines to the system."

```
reflect message to Network Process
print Network Process response
```

LOADBUF (D)

"Load a specified train image into either the 1403 universal character set buffer or the 3211 universal character set or form control buffers."

```
reflect message to URProcess
print URProcess response
```

NETWORK
[See MISCELLANEOUS FUNCTIONS]

VARY (B)

"Allow or disallow the availability of a device to a virtual machine or the VM/370 control program."

```
map raddr to device type
if device type is unit record:
   reflect message to URProcess
   print URProcess response
else:
   reflect message to Authorization Process
   print Authorization Process response
```

OPERATOR COMMANDS: PROGRAM ANALYSIS AND MONITOR FUNCTIONS

ACNT (A)

"Render accounting information for and to the user."

```
user ids - map each user id to NKCP id
           reflect message to each concerned NKCP
           gather responses
           print amalgamated response

ALL -      reflect message to each NKCP
           gather responses
           print amalgamated response
```

DCP (C,E)

"Display real processor storage locations."

[extend syntax to provide NKCP id (security level)]

reflect message to NKCP
print NKCP response

DMCP (C,E)

"Dump the real storage locations to a user's virtual printer."

[extend syntax to provide NKCP id (security level)]

reflect message to NKCP
print NKCP response

LOCATE (C,E)

"Provide the starting location of the user's CP control blocks or (virtual or real) devices."

```
user id - map user id to NKCP id
          reflect message to NKCP
          print NKCP response

raddr -   map raddr to device type
          if device type is unit record:
             reflect message to URProcess
             print URProcess response
          else:
             reflect message to Authorization Process
             print Authorization Process response
```

MONITOR
[See MISCELLANEOUS FUNCTIONS]

SAVESYS (E)

"Provide a storage copy of virtual machine storage, registers, and PSW contents as they currently exist."

not allowed as an operator command in KVM/370

STCP (C)

"Change the contents of real processor storage."

[extend syntax to provide NKCP id (security level)]

```
reflect message to NKCP
print NKCP response
```

Authorization Process
- Responses -

UserIdMapped

ResponseToOpRequest

NKCP Requests

OperatorMessage

QueryLogMsg

ResponseToOpRequest

Other Processes
- Responses -

ResponseToOpRequest

Other Processes
- Requests -

OperatorMessage

3.1.2: Operator Process Formal Specification

```
module OpProcess
type
DeviceTypes,
Char,
String = list of Char,

MessageLabel = (ResponseToOpRequest,OperatorRequest,MappedUserId,
          AddNkcp,DeleteNkcp,OpMsgPrinted,OpHitAttn,OpRequestRead),

KernelFunction = (GrantAccess,ReleaseDevice,CreateProcess,
          DestroyProcess,CreateAddressSpace,DestroyAddressSpace,
          RequestIO,ReceiveInterrupts,ReleaseCPU,ReceiveMessage),

ProcessName,
MessageId,
CommandName = (DIAL,LOGON,SET←RECORD,SET←MODE,SLEEP,
          UNLOCK←SYSTEM,UNLOCK←VIRT,ATTACH←CHANNEL,DETACH←CHANNEL,
          DISCONN,SAVESYS,
          AUTOLOG, INDICATE←IO,INDICATE←PAGING,
          LOGOFF,LOCK←SYSTEM,MONITOR←NKCP,NETWORK,QUERY←PAGING,
          QUERY←SASSIST←NKCP,QUERY←DASD,QUERY←TAPES,QUERY←LINES,
          QUERY←GRAF,QUERY←SYSTEM,QUERY←NAMES,QUERY←USERS,QUERY←USERID,
          QUERY←UR,QUERY←DUMP,QUERY←FILES,QUERY←READER,QUERY←PRINTER,
          QUERY←PUNCH,QUERY←HOLD,SET←SASSIST←NKCP,SET←DUMP,
          UNLOCK←SYSTEM,BACKSPAC,CHANGE,DRAIN,FLUSH,FREE,HOLD,ORDER,
          PURGE,REPEAT,SPACE,START,TRANSFER,DISABLE,ENABLE,LOADBUF,
          DCP,DMCP,STCP,
          FORCE←ALL,INDICATE←LOAD,INDICATE←QUEUES,MESSAGE←ALL,
          MONITOR←ALL,QUERY←SASSIST←ALL,QUERY←TDSK,SET←SASSIST←ALL,
          WARNING←ALL,ACNT←ALL,
          FORCE←USERID,INDICATE←USER,LOCK←USERID,MESSAGE←USERID,
          QUERY←PRIORITY,SET←FAVORED,SET←RESERVED,SET←PRIORITY,
          UNLOCK←USERID,WARNING←USERID,LOCATE←USERID,
          QUERY←RADDR,ATTACH←RADDR,DETACH←RADDR,VARY,LOCATE←RADDR,
          HALT,QUERY←STORAGE,
          ACNT←USERIDS,QUERY←LOGMSG,SET←LOGMSG,QUERY←ALL,SHUTDOWN),

Cat0 = T"(DIAL,LOGON,SET←RECORD,SET←MODE,SLEEP,
          UNLOCK←SYSTEM,UNLOCK←VIRT,ATTACH←CHANNEL,DETACH←CHANNEL,
          DISCONN,SAVESYS),

Cat1 = T"(AUTOLOG, INDICATE←IO,INDICATE←PAGING,
          LOGOFF,LOCK←SYSTEM,MONITOR←NKCP,NETWORK,QUERY←PAGING,
          QUERY←SASSIST←NKCP,QUERY←DASD,QUERY←TAPES,QUERY←LINES,
          QUERY←GRAF,QUERY←SYSTEM,QUERY←NAMES,QUERY←USERS,QUERY←USERID,
          QUERY←UR,QUERY←DUMP,QUERY←FILES,QUERY←READER,QUERY←PRINTER,
          QUERY←PUNCH,QUERY←HOLD,SET←SASSIST←NKCP,SET←DUMP,
          UNLOCK←SYSTEM,BACKSPAC,CHANGE,DRAIN,FLUSH,FREE,HOLD,ORDER,
          PURGE,REPEAT,SPACE,START,TRANSFER,DISABLE,ENABLE,LOADBUF,
          DCP,DMCP,STCP),
Cat2 = T"(FORCE←ALL,INDICATE←LOAD,INDICATE←QUEUES,MESSAGE←ALL,
        MONITOR←ALL,QUERY←SASSIST←ALL,QUERY←TOSK,SET←SASSIST←ALL,
        WARNING←ALL,ACNT←ALL),

Cat3 = T"(FORCE←USERID,INDICATE←USER,LOCK←USERID,MESSAGE←USERID,
        QUERY←PRIORITY,SET←FAVORED,SET←RESERVED,SET←PRIORITY,
        UNLOCK←USERID,WARNING←USERID,LOCATE←USERID),

Cat4 = T"(QUERY←RADDR,ATTACH←RADDR,DETACH←RADDR,VARY,LOCATE←RADDR),

Cat5 = T"(HALT,QUERY←STORAGE),

ConsoleOutputStatus = (Continuing,Idle),

ResponseStatus = (NoResponse, Responded),

RequestCategory = (OpRequest,ReadOpRequest,PrintOpMsg,MapUserId),

Answer = structure of(
        HMS = String,
        Text = String),

ResponseSlot = structure of(
        Respondent = ProcessName,
        Text = String,
        State = ResponseStatus),

PendingRequest = structure of(
        MsgId = MessageId,
        Kind = RequestCategory,
        Command = CommandName,
        Responses = set of ResponseSlot),

LogLine = structure of(
        Num = T"I:integer(1 <= I & I <= 99),
        Line = String)
```

```
variable
Answer: list of Answer,
PendingRequests: set of PendingRequest,
LogMessage: set of LogLine,
CommandExpected: boolean,
ConsoleOutputState: ConsoleOutputStatus,
CurrentNkcps: set of ProcessName
```

```
initial
Answers = nil
&
PendingRequests = Empty
&
LogMessage = Empty
&
CommandExpected = true
&
ConsoleOutputState = Idle
&
CurrentNkcps = Empty

invariant
A"P1,P2:PendingRequest (P1<:PendingRequests
                        &
                        P2<:PendingRequests ->
         (P1.MsgId = P2.MsgId -> P1 = P2))
&
A"P:PendingRequest,R1,R2:ResponseSlot
         (P<:PendingRequests&R1<:P.Responses&R2<:P.Responses ->
         (R1.Respondent = R2.Respondent -> R1 = R2))
&
A"L1,L2:LogLine(L1<:LogMessage
                &
                L2<:LogMessage ->
         (L1.Num = L2.Num -> L1 = L2))
```

```
constant
OpProcess, URProcess,AuthProcess,AcntProcess,UpdaterProcess:
        ProcessName,
IOPagingScheduler,IOScheduler,NetworkProcess,DumpProcessor:
        ProcessName,
TrustedProcesses = S"(OpProcess,URProcess,AuthProcess,
        AcntProcess,UpdaterProcess),
MsgName(String): MessageLabel,
Destination(CommandName): ProcessName,
DeviceType(DeviceAddress): DeviceTypes,
Raddr(String): DeviceAddress,
ClockRead: String,
ErrorMsg: String,
NewMsgId: MessageId,
SendMessage(ProcessName): KernelFunction,
KFcn(CommandName): KernelFunction,
DefinedElsewhere: boolean = true,
MakeString(set of LogLine): String,
LogPrompt: String,
Line#(String): T"I:integer(1 <= I & I <= 99),
Line(String): String

transform KernelCalled(K: KernelFunction)
effect (K = SendMessage => DefinedElsewhere
                 <> K = GrantAccess => DefinedElsewhere
                 <> K = ReleaseDevice => DefinedElsewhere
                 <> K = CreateProcess => DefinedElsewhere
                 <> K = DestroyProcess => DefinedElsewhere
                 <> K = CreateAddressSpace => DefinedElsewhere
                 <> K = DestroyAddressSpace => DefinedElsewhere
                 <> K = RequestIO => DefinedElsewhere
                 <> K = ReceiveInterrupts => DefinedElsewhere
                 <> K = ReleaseCPU => DefinedElsewhere
                 <> K = ReceiveMessage => DefinedElswhere
                 <> true)
```

```
transform OpCat0(Command: CommandName)

refcond Command<:Cat0

effect N"Answers = Answers;.(ClockRead, ErrorMsg)
          &
          N"CommandExpected
```

```
transform OpCatl(Command: CommandName)

refcond CommandExpected
        &
        Command<:Catl

effect N"PendingRequests =
        (Command = SET←DUMP =>
           PendingRequests
        <> PendingRequests || S"((NewMsgId,
                                  OpRequest,
                                  Command,
                                  S"((Destination(Command),
                                      nil,
                                      NoResponse)))))
        &
        N"CommandExpected
        &
        KernelCalled(SendMessage(Destination(Command)))
```

```
transform OpCat2(Command: CommandName)

refcond CommandExpected
        &
        Command<:Cat2

effect  N"CommandExpected
        &
        A"P:ProcessName(P<:CurrentNkcps ->
                KernelCalled(SendMessage(P)))
        &
        (Command ~<:S"(FORCE←ALL,WARNING←ALL) =>
                N"PendingRequests = PendingRequests ||
                  S"((NewMsgId,
                      OpRequest,
                      Command,
                      S"R:ResponseSlot(E"P:ProcessName
                        (P<:CurrentNkcps
                         &
                         R = (P,nil,NoResponse)))))
                <> N"PendingRequests = PendingRequests)
```

```
transform OpCat3(Command: CommandName,
                 Nkcps: set of ProcessName)

refcond Command<:Cat3

effect (Nkcps<<=CurrentNkcps =>
            N"PendingRequests =
              (Command<:S"(FORCE←USERID,
                           MESSAGE←USERID,
                           WARNING←USERID) => PendingRequests
            <> PendingRequests ||
             S"((NewMsgId,
                 OpRequest,
                 Command,
                 S"R:ResponseSlot(E"P:ProcessName
                      (P<:Nkcps
                       &
                       R = (P,nil,NoResponse))))))
             &
             N"CommandExpected
             &
             A"P:ProcessName(P<:Nkcps ->
                 KernelCalled(SendMessage(P)))
             &
             NoError
          <> Error
             &
             N"PendingRequests = PendingRequests
             &
             N"CommandExpected)
```

```
transform OpCat4(Command: CommandName,
                 Raddr: DeviceAddress)

refcond CommandExpected
        &
        Command<:Cat4

effect  N"PendingRequests = PendingRequests ||
          S"((NewMsgId,
              OpRequest,
              Command.
              S"(((DeviceType(Raddr) = UnitRecord =>
                URProcess
             <> AuthProcess),
                  nil,
                  NoResponse))))
        &
        N"CommandExpected
        &
        (DeviceType(Raddr) = UnitRecord =>
                KernelCalled(SendMessage(URProcess))
          <> KernelCalled(SendMessage(AuthProcess)))
```

```
transform OpCat5(Command: CommandName)

refcond CommandExpected
        &
        Command<:Cat5

effect  KernelCalled(KFcn(Command))
        &
        N"Answers = Answers;.(ClockRead,KernelResult)
        &
        N"CommandExpected
```

```
transform OpCat6(Command: CommandName)

refcond CommandExpected
        &
        Command<:Cat3 || S"(ACNT←USERIDS)

effect  N"PendingRequests = PendingRequests ||
          S"((NewMsgId,
               MapUserId,
               Command,
               S"((AuthProcess,nil,NoResponse))))
        &
        N"CommandExpected
        &
        KernelCalled(SendMessage(AuthProcess))
```

```
transform OpCat7(Command: CommandName,
                 Nkcps: set of ProcessName)

refcond Command = ACNT-USERIDS

effect  (Nkcps<<=CurrentNkcps =>
                N"PendingRequests = PendingRequests ||
                  S"((NewMsgId,
                      OpRequest,
                      Command,
                      S"R:ResponseSlot(E"P:ProcessName
                        (P<:Nkcps
                         &
                         R = (P,nil,NoResponse)))))
                &
                N"CommandExpected
                &
                A"P:ProcessName(P<:Nkcps ->
                        KernelCalled(SendMessage(P)))
                &
                NoError
         <> Error
            &
            N"PendingRequests = PendingRequests
            &
            N"CommandExpected)
```

```
transform OpCat8a(Command: CommandName)

refcond CommandExpected
        &
        Command = QUERY←LOGMSG

effect  N"Answers = Answers:.(ClockRead,MakeString(LogMessage))
        &
        N"CommandExpected
```

```
transform OpCat8b(Command: CommandName)

refcond CommandExpected
        &
        Command = SET-LOGMSG

effect  N"Answers = Answers:.(ClockRead,LogPrompt)
        &
        N"CommandExpected = false
```

```
transform OpCat8c(Line#: T"]:integer(1 <= ] & ] <= 99),
                 Line: String)

refcond ~CommandExpected

effect  N"CommandExpected
        &
        (E"L:LogLine(L<:LogMessage &
           L.Num = Line#) =>
                E"L:LogLine(L<:LogMessage &
                   L.Num = Line#
                   &
                   N"LogMessage = LogMessage ~~ S"(L) ||
                        S"((Line#.Line))
         <> N"LogMessage = LogMessage || S"((Line#.Line)))
```

```
transform OpCat9(Command: CommandName)

refcond CommandExpected
        &
        Command = QUERY←ALL

effect  N"CommandExpected
        &
        N"PendingRequests = PendingRequests ||
          S"((NewMsgId,
              Command,
              S"((Kernel,KernelResult,Responded),
                 (AuthProcess,nil,NoResponse),
                 (URProcess,nil,NoResponse))))
        &
        KernelCalled(KFcn(QUERY←STORAGE))
        &
        KernelCalled(SendMessage(AuthProcess))
        &
        KernelCalled(SendMessage(URProcess))
```

```
transform OpCat10(Command: CommandName)

refcond CommandExpected
        &
        Command = SHUTDOWN

effect  N"CommandExpected
        &
        KernelCalled(KFcn(SHUTDOWN))
        &
        KernelCalled(SendMessage(AuthProcess))
        &
        KernelCalled(SendMessage(URProcess))
        &
        KernelCalled(SendMessage(AcntProcess))
        &
        A"P:ProcessName(P<:CurrentNkcps ->
                KernelCalled(SendMessage(P)))
```

```
transform AUTH1(Nkcp: ProcessName)

effect  (Nkcp<:CurrentNkcps =>
                Error
                &
                N"CurrentNkcps = CurrentNkcps
          <> NoError
             &
             N"CurrentNkcps = CurrentNkcps || S"(Nkcp))
```

```
transform AUTH2(Nkcp: ProcessName)

effect  (Nkcp<:CurrentNkcps =>
                 NoError
                 &
                 N"CurrentNkcps = CurrentNkcps ~~ S"(Nkcp)
         <>      Error
                 &
                 N"CurrentNkcps = CurrentNkcps)
```

```
transform ProcessedResponse(P: PendingRequest,
                  Text: String,
                  Source: ProcessName)

refcond P<:PendingRequests
        &
        P.Kind = OpRequest
        &
        MsgName(Text) = ResponseToOpRequest

effect  (E"R:ResponseSlot(R<:P.Responses &
           R.Respondent = Source
           &
           R.State = NoResponse) =>
              E"R:ResponseSlot(R<:P.Responses &
                R.Respondent = Source
                &
                R.State = NoResponse
                &
                (E"R1:ResponseSlot(R1<:P.Responses &
                   R1 ~= R
                   &
                   R1.State = NoResponse) =>
                        N"Answers = Answers
                        &
                        N"PendingRequests = PendingRequests ~~ S"(P) ||
                            S"((P.MsgId,
                                P.Kind,
                                P.Command,
                                P.Responses ~~ S"(R) ||
                                    S"((R.Respondent,
                                        Response(Text),
                                        Responded))))
                  <>    N"PendingRequests = PendingRequests ~~ S"(P)
                        &
                        N"Answers = Answers:.OpResponse(P)))
         <> Error)
```

```
transform ProcessedCommand(Text: String)

effect  (CommandExpected =>
            E"Command:CommandName(Command = CommandName(Text) &
             (Command<:Cat0 =>
                          OpCat0(Command)
              <> Command<:Cat1 =>
                          OpCat1(Command)
              <> Command<:Cat2 =>
                          OpCat2(Command)
              <> Command<:Cat3 =>
                          OpCat6(Command)
              <> Command<:Cat4 =>
                          OpCat4(Command,Raddr(Text))
              <> Command<:Cat5 =>
                          OpCat5(Command)
              <> Command = ACNT←USERIDS =>
                          OpCat6(Command)
              <> Command = QUERY←LOGMSG =>
                          OpCat8a(Command)
              <> Command = SET←LOGMSG =>
                          OpCat8b(Command)
              <> Command = QUERY←ALL =>
                          OpCat9(Command)
              <> Command = SHUTDOWN =>
                          OpCat10(Command)
              <> Error))
        <> OpCat8c(Line#(Text),Line(Text)))
```

```
transform MsgNkcp(MsgId: MessageId,
                  Text: String,
                  Source: ProcessName)

refcond Source ~<:TrustedProcesses ||
                  S"(IOPagingScheduler,IOScheduler,
                     NetworkProcess,DumpProcessor)

effect   (E"P:PendingRequest(P<:PendingRequests &
            P.MsgId = MsgId) =>
               E"P:PendingRequest(P<:PendingRequests &
                 P.MsgId = MsgId
                 &
                 (P.Kind = OpRequest
                 &
                 MsgName(Text) = ResponseToOpRequest =>
                    ProcessedResponse(P,Text,Source)
                  <> Error))
         <> (MsgName(Text) = OperatorRequest =>
                 N"Answers = Answers;.(ClockRead,Text)
              <> Error))
```

```
transform MsgUR(MsgId: MessageId,
                Text: String,
                Source: ProcessName)

refcond Source = URProcess

effect  (E"P:PendingRequest(P<:PendingRequests &
            P.MsgId = MsgId) =>
               E"P:PendingRequest(P<:PendingRequests &
                 P.MsgId = MsgId
                 &
                 (P.Kind = OpRequest
                 &
                 MsgName(Text) = ResponseToOpRequest =>
                    ProcessedResponse(P,Text,Source)
                  <> Error))
          <> (MsgName(Text) = OperatorRequest =>
                 N"Answers = Answers;.(ClockRead,Text)
              <> Error))
```

```
transform MsgIOPS(MsgId: MessageId,
                  Text: String,
                  Source: ProcessName)

refcond Source = IOPagingScheduler

effect  (E"P:PendingRequest(P<:PendingRequests &
           P.MsgId = MsgId) =>
              E"P:PendingRequest(P<:PendingRequests &
                P.MsgId = MsgId
                &
                (P.Kind = OpRequest
                &
                MsgName(Text) = ResponseToOpRequest =>
                   ProcessedResponse(P,Text,Source)
                 <> Error))
         <> (MsgName(Text) = OperatorRequest =>
                N"Answers = Answers:.(ClockRead,Text)
             <> Error))
```

```
transform MsgIOS(MsgId: MessageId,
                 Text: String,
                 Source: ProcessName)

refcond Source = IOScheduler

effect   (E"P:PendingRequest(P<:PendingRequests &
              P.MsgId = MsgId) =>
                 E"P:PendingRequest(P<:PendingRequests &
                   P.MsgId = MsgId
                   &
                   (P.Kind = OpRequest
                   &
                   MsgName(Text) = ResponseToOpRequest =>
                      ProcessedResponse(P,Text,Source)
                    <> Error))
           <> (MsgName(Text) = OperatorRequest =>
                   N"Answers = Answers:.(ClockRead,Text)
               <> Error))
```

```
transform MsgDump(MsgId: MessageId,
                  Text: String,
                  Source: ProcessName)

refcond Source = DumpProcessor

effect  (E"P:PendingRequest(P<:PendingRequests &
           P.MsgId = MsgId) =>
              E"P:PendingRequest(P<:PendingRequests &
                P.MsgId = MsgId
                &
                (P.Kind = OpRequest
                &
                MsgName(Text) = ResponseToOpRequest =>
                   ProcessedResponse(P,Text,Source)
                 <> Error))
          <> (MsgName(Text) = OperatorRequest =>
                N"Answers = Answers:.(ClockRead,Text)
            <> Error))
```

```
transform MsgAcnt(MsgId: MessageId,
                  Text: String,
                  Source: ProcessName)

refcond Source = AcntProcess

effect  (E"P:PendingRequest(P<:PendingRequests &
           P.MsgId = MsgId) =>
              E"P:PendingRequest(P<:PendingRequests &
                P.MsgId = MsgId
                &
                (P.Kind = OpRequest
                &
                MsgName(Text) = ResponseToOpRequest =>
                   ProcessedResponse(P,Text,Source)
                 <> Error))
         <> (MsgName(Text) = OperatorRequest =>
                N"Answers = Answers:.(ClockRead,Text)
            <> Error))
```

```
transform MsgAuth(MsgId: MessageId,
              Text: String,
              Source: ProcessName)

refcond Source = AuthProcess

effect   (E"P:PendingRequest(P<:PendingRequests &
            P.MsgId = MsgId) =>
                E"P:PendingRequest(P<:PendingRequests &
                   P.MsgId = MsgId
                   &
                   (P.Kind = OpRequest =>
                       (MsgName(Text) = ResponseToOpRequest =>
                              ProcessedResponse(P,Text,Source)
                        <> Error)
                    <> P.Kind = MapUserId =>
                          (MsgName(Text) = MappedUserId =>
                              (P.Command<:Cat3 =>
                                         OpCat3(P.Command,Nkcps(Text))
                               <> P.Command = ACNT←USERIDS =>
                                         OpCat7(P.Command,Nkcps(Text))
                               <> Error)
                           <> Error)
                    <> Error))
         <> (MsgName(Text) = OperatorRequest =>
                              N"Answers = Answers:.(ClockRead,Text)
               <> MsgName(Text) = AddNkcp =>
                              AUTH1(Nkcps(Text))
               <> MsgName(Text) = DeleteNkcp =>
                              AUTH2(Nkcps(Text))
               <> Error))
```

```
transform MsgNet(MsgId: MessageId,
                 Text: String,
                 Source: ProcessName)

refcond Source = NetworkProcess

effect  (E"P:PendingRequest(P<:PendingRequests &
           P.MsgId = MsgId) =>
              E"P:PendingRequest(P<:PendingRequests &
                P.MsgId = MsgId
                &
                (P.Kind = OpRequest =>
                   (MsgName(Text) = ResponseToOpRequest =>
                        ProcessedResponse(P.Text,Source)
                    <> Error)
                 <> P.Kind = PrintOpMsg =>
                     (MsgName(Text) = OpMsgPrinted =>
                                (Answers = nil =>
                                   N"ConsoleOutputState = Idle
                                 <> KernelCalled(SendMessage(
                                        NetworkProcess))
                                    &
                                    N"ConsoleOutputState = Continuing
                                    &
                                    N"Answers = Answers:2)
                   <> MsgName(Text) = OpHitAttn =>
                          N"CommandExpected
                          &
                          N"ConsoleOutputState = Idle
                          &
                          KernelCalled(SendMessage(NetworkProcess))
                    <> Error)
                <> P.Kind = ReadOpRequest =>
                   (MsgName(Text) = OpRequestRead =>
                          ProcessedCommand(Text)
                          &
                          (CommandExpected
                           &
                           ConsoleOutputState = Idle
                           &
                           Answers ~= nil =>
                               KernelCalled(SendMessage(
                                       NetworkProcess))
                               &
                               N"Answers = Answers:2
                               &
                               N"ConsoleOutputState = Continuing
                           <> N"Answers = Answers
                               &
                               N"ConsoleOutputState =
                                       ConsoleOutputState)
```

```
                      <> MsgName(Text) = OpHitAttn =>
                            N"CommandExpected
                            &
                            N"ConsoleOutputState = Idle
                            &
                            KernelCalled(SendMessage(NetworkProcess))
                      <> Error)
                  <> Error))
      <> (MsgName(Text) = OperatorRequest =>
                         N"Answers = Answers:.(ClockRead,Text)
          <> MsgName(Text) = OpHitAttn =>
                         N"CommandExpected
                         &
                         N"ConsoleOutputState = Idle
                         &
                         KernelCalled(SendMessage(NetworkProcess))
          <> Error))
```

```
transform OpDriver(InterruptType:,
                   InterruptSubType:,
                   MsgId: MessageId,
                   Text: String,
                   Source: ProcessName)

effect  (InterruptType = ExternalInterrupt =>
           (InterruptSubType = Message =>
                KernelCalled(ReceiveMessage)
                &
                (Source = AuthProcess =>
                        MsgAuth(MsgId,Text,Source)
                 <> Source = URProcess =>
                        MsgUR(MsgId,Text,Source)
                 <> Source = IOPagingScheduler =>
                        MsgIOPS(MsgId,Text,Source)
                 <> Source = IOScheduler =>
                        MsgIOS(MsgId,Text,Source)
                 <> Source = NetworkProcess =>
                        MsgNet(MsgId,Text,Source)
                 <> Source = DumpProcessor =>
                        MsgDump(MsgId,Text,Source)
                 <> Source~<:TrustedProcesses ||
                        S"(IOPagingScheduler,IOScheduler,
                           NetworkProcess,DumpProcessor) =>
                        MsgNkcp(MsgId,Text,Source)
                 <> Error)
            <> Error)
         <> Error)
        &
        KernelCalled(ReceiveInterrupts)
        &
        KernelCalled(ReleaseCPU)
end OpProcess
```

3.2.1: Unit Record Process Informal Description

This section contains the informal description of the Unit Record Process of KVM/370.

Overview

The Unit Record Process (URProcess) performs four main functions:

(1) routes readers to the appropriate NKCP input spool file processor;

(2) controls the allocation of printers and punches to NKCP output spool file processors;

(3) processes operator commands dealing with spool files or unit record devices; and

(4) controls the allocation of all unit record devices and tapes.

Input Spool Files

The Unit Record Process receives an external interrupt from the Kernel which tells the Unit Record Process that a particular card reader is ready to transfer card images. The Unit Record Process initiates a single-card read, which causes the identification card to be read. This identification card contains (at least) the security level of the information in the file. The Unit Record Process attaches the device to the appropriate NKCP and sends a message to the NKCP controlling this security level, notifying the NKCP that an input file awaits spooling. When the NKCP's input spool file processor has completed the transfer it notifies the Unit Record Process, which in turn detaches the input device.

Output Spool Files

When the output spool file processor of an NKCP wishes to print or punch a (set of) spool file(s), it sends a message to that effect to the Unit Record Process, which in turn enters the NKCP in a list of NKCPs awaiting a real device. When an appropriate device becomes available, the Unit Record Process causes the device to be readied for the new security level (possibly by the operator, according to installation policy), attaches the device to the NKCP, and notifies the NKCP. When the NKCP is finished with the device, it notifies the Unit Record Process, which detaches the device from the NKCP. The Unit Record Process then marks the device as available, allowing the cycle to begin anew.

Operator Commands

Operator commands regarding spool files or unit record devices are sent by the Operator Process to the Unit Record Process for processing. Most information about spool files is distributed in the NKCPs. Thus the Unit Record Process processes operator commands dealing with spool files by sending messages to the NKCPs and awaiting their responses. It then gathers the responses and prepares one message for the Operator Process, which will cause the response to be printed on the operator's console.

Each operator command processed by the Unit Record Process is discussed below. Figures 1 through three display the various processing cycles of the unit record devices.

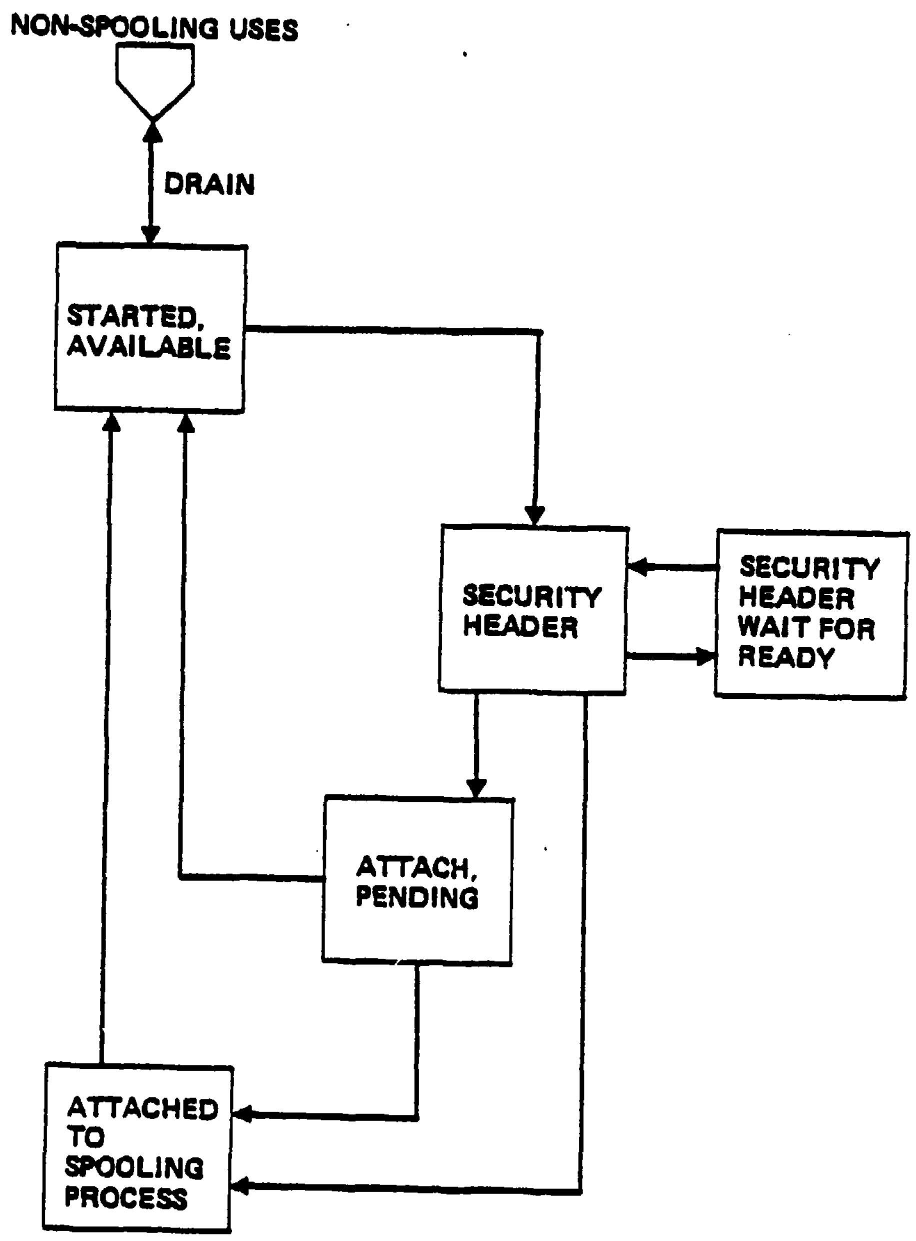

Figure 1

THE READER SPOOLING CYCLE

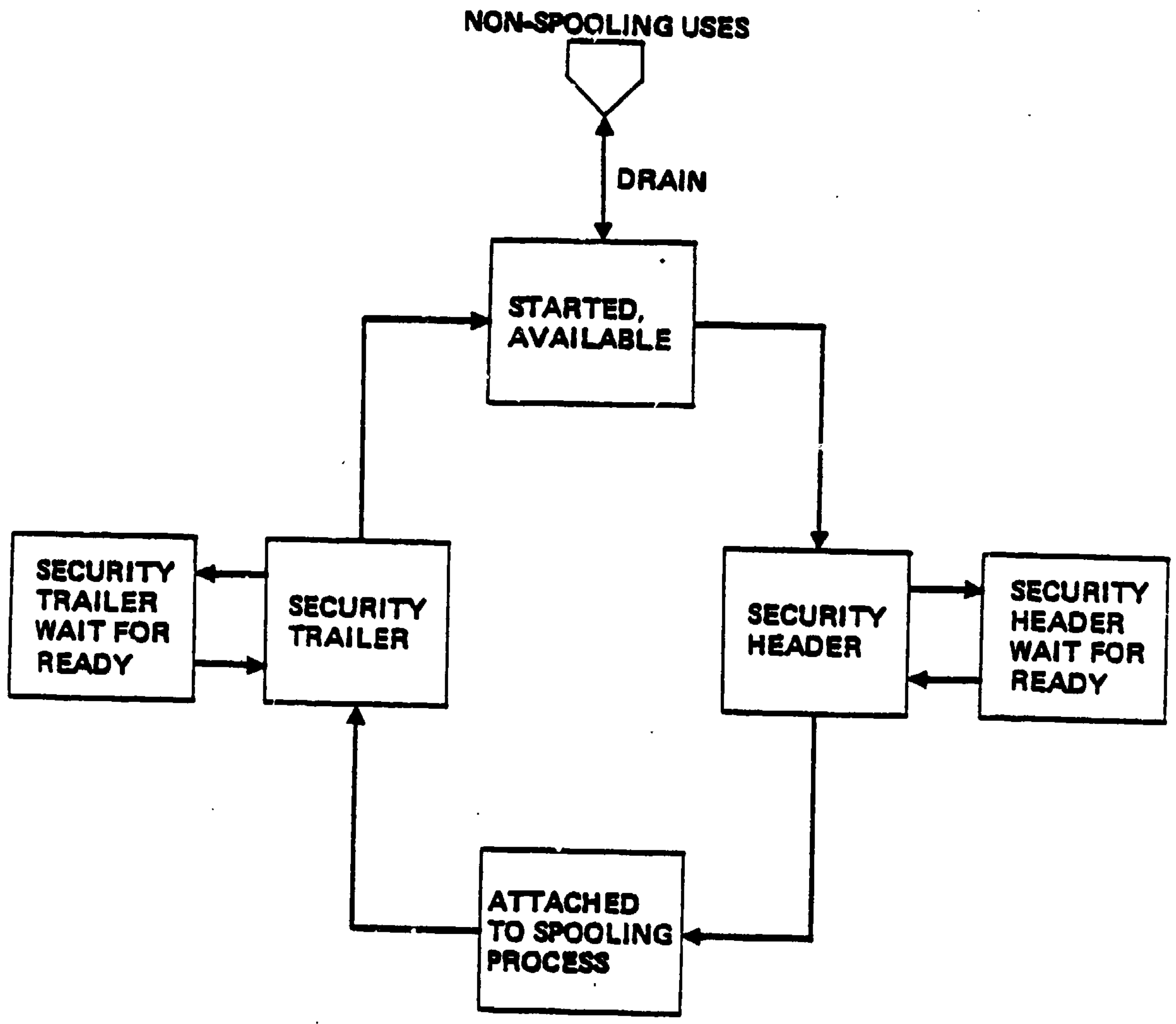
NON-SPOOLING USES
DRAIN
STARTED, AVAILABLE
SECURITY HEADER
SECURITY HEADER WAIT FOR READY
ATTACHED TO SPOOLING PROCESS
SECURITY TRAILER
SECURITY TRAILER WAIT FOR READY

Figure 2

Printer and Punch

Spooling Cycles

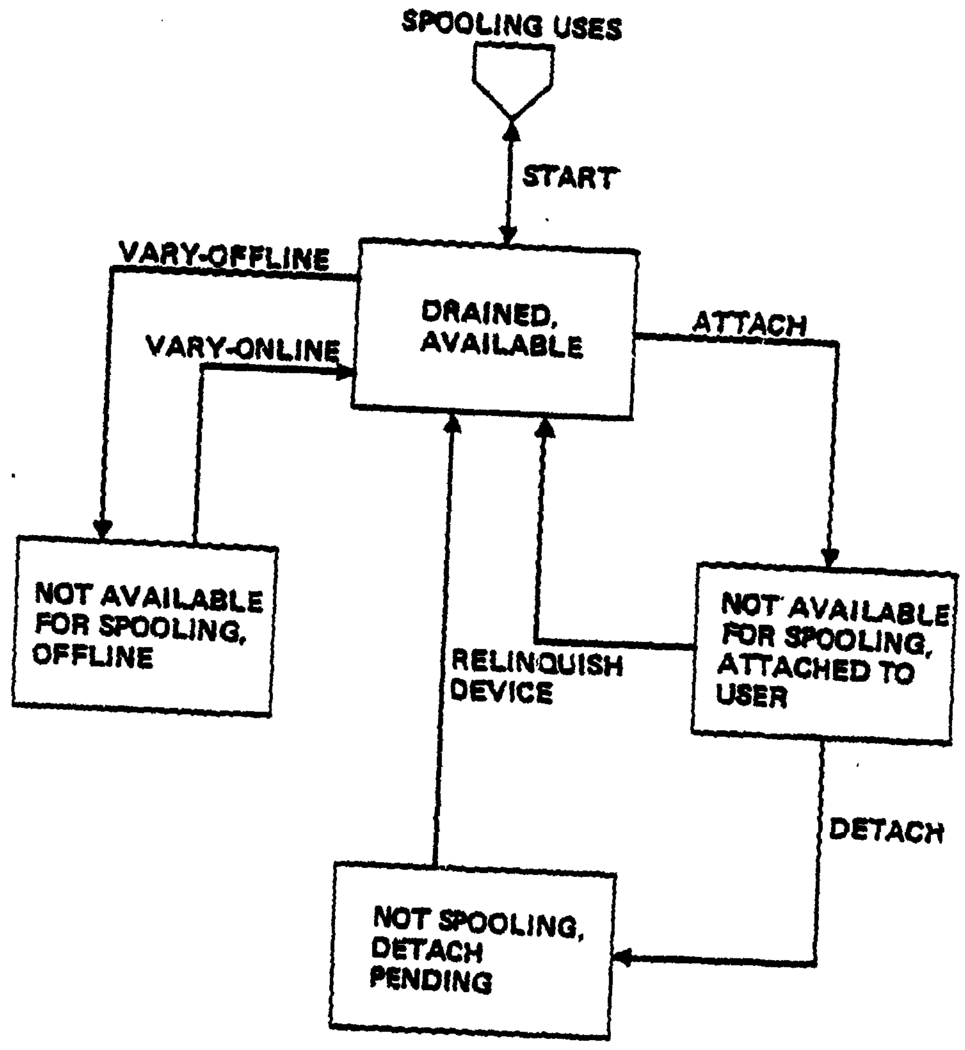
SPOOLING USES
START
VARY-OFFLINE
VARY-ONLINE
DRAINED, AVAILABLE
ATTACH
NOT AVAILABLE FOR SPOOLING, OFFLINE
RELINQUISH DEVICE
NOT AVAILABLE FOR SPOOLING, ATTACHED TO USER
DETACH
NOT SPOOLING, DETACH PENDING

Figure 3

NON-SPOOLING CYCLES FOR ALL UNIT RECORD DEVICES

Operator Commands
sent first to Unit Record Process

Command	S/M	Map	Cate-gory	Destination(s)
QUERY←UR	Both	-	0	Attached Processes
QUERY←ALL	Both	-	0	Attached Processes
QUERY←RADDR	Both	-	0	Attached Processes
QUERY←TAPES	Both	-	0	Attached Processes
QUERY←FILES←USERID	S	U->N	3	NKCP
QUERY←FILES←ALL	M	-	2	all NKCPs
QUERY←READER←USERID	S	U->N	3	NKCP
QUERY←READER←SPOOLID	S	-	1	NKCP
QUERY←READER←ALL	M	-	2	all NKCPs
QUERY←PRINTER←USERID	S	U->N	3	NKCP
QUERY←PRINTER←SPOOLID	S	-	1	NKCP
QUERY←PRINTER←ALL	M	-	2	all NKCPs
QUERY←PUNCH←USERID	S	U->N	3	NKCP
QUERY←PUNCH←SPOOLID	S	-	1	NKCP
QUERY←PUNCH←ALL	M	-	2	all NKCPs
QUERY←HOLD	M	-	2	all NKCPs
SHUTDOWN	See command description			
BACKSPAC	S	-	4	Attached Process
CHANGE←USERID	S	U->N	3	NKCP
CHANGE←SYSTEM←CLASS←ALL	M	-	2	all NKCPs
CHANGE←SYSTEM←SPOOLID	S	-	1	NKCP
DRAIN	See command description			
FLUSH	S	-	4	Attached Process
FREE	S	U->N	3	NKCP
HOLD	S	U->N	3	NKCP
ORDER←USERID	S	U->N	3	NKCP
ORDER←SYSTEM	Both	-	6	some NKCPs
PURGE←USERID	S	U->N	3	NKCP
PURGE←SYSTEM	Both	-	6	some NKCPs
REPEAT	S	-	4	Attached Process
SPACE	S	-	5	Attached Process
START	See command description			
TRANSFER	See command description			
ATTACH	See command description			
DETACH	See command description			
LOADBUF	See command description			
VARY	See command description			
LOCATE	S	-	7	Attached Process

Dedicated Device Attachment

The Unit Record Process also oversees the assignment of unit record devices and tapes in nonsharable status to VMs. There are two ways a user can initiate the attachment of a dedicated device. At logon time, the Authorization Process will automatically request device attachment if a permanent device attachment request exists in the user's directory entry. The user may also request device attachment from the operator who will send the request on to the Unit Record Process. In either case, the device must be available for assignment (i.e., not attached to some other process and in the case of readers, printers, and punches, not involved in spooling operations) and must meet security requirements. The Unit Record Process performs the attachment if it can and notifies the requesting process of the result. (Further details can be found in a later section.)

More on Readers

Each reader's initial state is either "Drained, Available" or "NotAvailableForSpooling, OffLine," depending on the system configuration information provided by the Authorization Process.

The "Drained, Available" state provides the link between the spooling and nonspooling cycles: to be employed in a nonspooling capacity, a device must first reach this state. For example, the DRAIN command must be issued for a device before it can be attached to a user, or varied offline.

A device can be entered into the spooling cycle by issuing the START command. The reader's associated state then becomes "Started, Available." The very first time the START command is issued, a list of file classes the device may process must be provided.

A Walk Around the Reader Spooling Cycle

The operator first issues the START command, providing a list of file classes which the reader may process.

The operator then loads a deck of cards into the reader's hopper and presses the read button. The resulting "Ready" interrupt ("Device End") causes a state transition from "Started, Available" to "Started, SecurityHeader". The Unit Record Process initiates a card read operation to read in the header card that contains the security level of the deck.

The interrupt generated by the end of the card causes the next state change. If there is no read error, the Unit Record Process attaches the device to the NKCP which should process this security level. The state changes to "Started, SpoolingForProcess." If there is a read error, the Unit Record Process notifies the operator that physical intervention is required by sending a message to the Operator Process. The Unit Record Process then updates its state information for this reader to reflect that a wait for a "Device Ready" interrupt is necessary. Upon receipt of this interrupt (caused by the operator again pushing the read button), the Unit Record Process reissues its request for input of the header card.

If the appropriate NKCP does not currently exist, the Unit Record Process asks the Authorization Process to create it. The state becomes "Started, AttachPending." When the reader has reached this state, further processing awaits the receipt of a message from the Authorization Process stating whether or not the NKCP was created.

Once the reader is under the control of the NKCP, the Unit Record Process updates the status of the reader to "Started, SpoolingForProcess." The Unit Record Process pays no more attention to the reader until the controlling NKCP explicitly requests its release.

For various reasons the Unit Record Process can reject the request that the reader be attached to an NKCP. These reasons include:

- o the reader is not cleared to read this deck (operator error); or
- o the NKCP which should read this deck does not currently exist, and for some reason the Authorization Process is not going to create it.

In these cases, the Unit Record Process returns the device to the "Available" state and sends a message to the Operator Process asking the operator to physically flush the deck from the reader.

If the attach succeeds and the card deck is processed by some NKCP, the Unit Record Process will next receive a release request from the NKCP. At this point, the Unit Record Process detaches the device from the NKCP, and updates the device status to "Started, Available". The cycle is then complete.

A More Detailed View of Dedicated Device Attachment

At logon, the Authorization Process attempts to attach to the new VM each of the devices listed in the dedicated device section of the directory entry for the user logging on. (See the description of the Authorization Process for a detailed view of the directory and its contents.) If the device being requested is either a unit record device or a tape drive, the Authorization Process, after examining the security conditions of the attachment, requests that the actual assignment be performed by the Unit Record Process. The Unit Record Process determines if the device is available (not attached to any other process, and if a unit record device, drained), and attaches the device if it can. It informs the Authorization Process of the result. The Unit Record Process does not directly notify the NKCP, since the Authorization Process will inform the NKCP of the device attachment at the same time it tells the NKCP about the new VM.

When the NKCP is finished with the device (because the user performed an explicit detach or because he or she logged off), the NKCP is expected to directly notify the Unit Record Process which will update its status information and detach the device from the NKCP.

The other method of attaching a device for nonsharable use requires that the user send a message to the system operator who decides if such an attachment should be performed. The operator first ensures the device is available and/or drained, and then types the operator command 'Attach' with the appropriate parameters. If the device is a unit record device or a tape drive, the Operator Process reflects the message to the Unit Record Process. The Unit Record Process determines whether the device is available and/or drained, and whether or not the process and device have compatible security levels given the requested access. If these conditions are met, the Unit Record Process performs the attachment. If the attachment is performed, the Unit Record Process informs both the operator and the affected NKCP. If it cannot be performed, the Unit Record Process informs only the operator. If the operator then decides that the device is not currently assignable to the NKCP, it is up to him or her to so inform the NKCP. Again, when the NKCP is finished with the device, it directly informs the Unit Record Process which releases the assignment.

In summary, dedicated unit record or tape drive device attachment is performed either at logon time or by operator request, rather than by the user directly requesting attachment from the Unit Record Process. However, device detachment is directly requested of the Unit Record Process.

OPERATOR PROCESS
- Requests -

QUERY

"Provide status information on the real or virtual machine, and miscellaneous CP functions."

```
UR,'ALL - for each device in each device table:
             if device is currently attached to NKCP:
                reflect message to NKCP
                prepare response from status information
                   and NKCP response
             else:
                prepare response from status information

             send response to Operator Process

raddr -   [raddr must be unit record device]

          if device is currently attached to NKCP:
             reflect message to NKCP
             prepare response from status information
                and NKCP response
          else:
             prepare response from status information

          send response to Operator Process

FILES -   user id -  map user id to NKCP id
                     reflect message to NKCP
                     send NKCP response to Operator Process

          [ALL]   -  reflect message to each NKCP
                     gather responses
                     send amalgamated response
                        to Operator Process

READER, PRINTER, PUNCH -
          user id -  map user id to NKCP id
                     reflect message to NKCP
                     send NKCP response to Operator Process

          spool id - [syntax has been extended
                      to provide NKCP ids along
                      with all spool ids]

                     reflect message to NKCP
                        owning spool file
                     send NKCP response to Operator Process
```

```
              [ALL] -    reflect message to each NKCP
                         gather responses
                         send amalgamated response
                            to Operator Process

        HOLD -    reflect message to each NKCP
                  gather responses
                  send amalgamated response to Operator Process
```

BACKSPAC

"Restart or backspace the output of a unit record spooling device."

```
        [device must be printer or punch;
         must be assigned to Process]

        reflect message to Process owning device
        send Process response to Operator Process
```

CHANGE

"Alter one or more attributes of a closed spooled file."

```
        user id - map user id to NKCP id
                  reflect message to NKCP
                  send NKCP response to Operator Process

        SYSTEM -  if "CLASS" or "ALL" specified:
                     reflect message to each NKCP
                     gather responses
                     send amalgamated response
                        to Operator Process
                  else: (spool id specified)
                        [syntax has been extended to
                         provide NKCP ids along with
                         all spool ids]

                     reflect message to NKCP owning spool file
                     send NKCP response to Operator Process
```

DRAIN

"Stop spooling activity on the specified devices after the current files in operation reach termination."

```
        update appropriate device table entries
        reflect message to each Process
          owning an affected device,
          if no message has already been sent
          gather responses
          send amalgamated response to Operator Process
```

FLUSH

"Eliminate and halt the current file output on the specified real unit record device."

```
          [device must be printer or punch;
           must be assigned to Process]

          reflect message to Process owning device
          send Process response to Operator Process
```

FREE

"Release spool HOLD status from previously held files for real reader, printer, and punch devices."

```
          map user id to NKCP id
          reflect message to NKCP
          send NKCP response to Operator Process
```

HOLD

"Defer file output to the real reader, printer, and punch devices."

```
          map user id to NKCP id
          reflect message to NKCP
          send NKCP response to Operator Process
```

ORDER

"Redefine the order of closed spool files."

```
          user id - map user id to NKCP id
                    reflect message to NKCP
                    send NKCP response to Operator Process

          SYSTEM -  determine concerned NKCPs
                    reflect message to each
                    gather responses
                    send amalgamated response to Operator Process
```

PURGE

"Remove spooled file(s) before reading, punching, or printing."

```
user id - map user id to NKCP id
          reflect message to NKCP
          send NKCP response to Operator Process

SYSTEM -  determine concerned NKCPs
          reflect message to each
          gather responses
          send amalgamated response to Operator Process
```

REPEAT

"Add to the number of copies of an output printer or punch file on a real unit record device."

```
[raddr must be a printer or punch;
 must be assigned to an NKCP]

reflect message to Process owning device
send Process response to Operator Process
```

SPACE

"Force single space on the printer, regardless of carriage control codes contained in the file."

```
[raddr must be a printer;
 must be assigned to an NKCP]

reflect message to Process owning device
send Process response to Operator Process
```

START

"Start unit record devices, or restart drained devices, or restart and change output classes that may be serviced."

```
update appropriate device table entry
send response to Operator Process
```

TRANSFER

"Direct an input spool file to or retrieve it from a specified user's virtual card reader."

```
[syntax has been extended to provide NKCP ids
 along with all user ids]
```

```
user id, SYSTEM spoolid -
     TO user id, FROM user id -
        if source NKCP = destination NKCP:
           reflect message to source NKCP
           send NKCP response to Operator Process
        else:
           send "Illegal" message to Operator Process

     FROM ALL - not allowed in KVM/370

SYSTEM ALL -
     TO user id - not allowed in KVM/370

     FROM user id - [same as SYSTEM spoolid FROM user id]
```

ATTACH

"Attach a real device to a virtual machine or the real system."

```
[raddr must be a unit record device:
 device must be in a DRAINED state]

KernelCall: GrantAccess
update appropriate device table entry
send response to Operator Process
```

DETACH

"Remove a real or virtual device or channel from a virtual machine or the real system."

```
[raddr must be a unit record device:
 device must be in a DRAINED state]

KernelCall: ReleaseDevice
update appropriate device table entry
send response to Operator Process
```

LOADBUF

"Load a specified train image into either the 1403 universal character set buffer or the 3211 universal character set or form control buffers."

```
[raddr must be a printer in a DRAINED state]

if device is currently attached to NKCP:
   reflect message to NKCP
   send NKCP response to Operator Process
```

```
        else:
           update appropriate device table entry
```

VARY

"Allow or disallow the availability of a device to the VM/370 control program."

```
        raddr - [raddr must be a unit record device]

                update appropriate device table entry
                if device is attached to Process, notify Process
                send response to Operator Process
```

LOCATE

"Provide the starting location of the user's CP control blocks or (virtual or real) devices."

```
        raddr - [raddr must be a unit record device;
                 device must be assigned to NKCP]

                reflect message to NKCP owning device
                send NKCP response to Operator Process
```

Authorization Process
- Responses -

AddedNkcp

CannotAddNkcp

UserIdMapped

Authorization Process
- Requests -

AttachDevice

DeleteNkcp

NKCP Requests

AssignSpoolDevice

ReleaseSpoolDevice

DetachDevice

NKCP Responses

ResponseToOpRequest

DetachDevice

3.2.2: Unit Record Process Formal Specification

```
module URProcess

type
DeviceAddress,
Char,
String = list of Char,

HardwareStatus,
MessageId,
ProcessName,
Class,
ResponseStatus = (NoResponse,Responded),

RelDevRequestStatus = (NoNeed,ShouldSend,Sent),

CommandName = (QUERY←UR←ALL,QUERY←RADDR,
        QUERY←FILES←USERID←,QUERY←FILES←ALL,
        QUERY←RDPRPU←USERID,QUERY←RDPRPU←SPOOLID,
        QUERY←RDPRPU←ALL,QUERY←HOLD,BACKSPAC,CHANGE←USERID,
        CHANGE←SYSTEM,DRAIN,FLUSH,FREE,HOLD,ORDER←USERID,
        ORDER←SYSTEM,PURGE←USERID,PURGE←SYSTEM,REPEAT,SPACE,
        TRANSFER,LOCATE←RADDR),

RequestCategory = (OpRequest,MapUserId,NeedNkcp,RelinquishDevice),

InputDeviceStatus = (SecurityHeader,SecurityHeaderWaitForReady,
        AttachPending,AttachedToSpoolingProcess,Available,
        AttachedToUser,DetachPending,OffLine),

OutputDeviceStatus = (SecurityHeader,SecurityHeaderWaitforReady,
        AttachedToSpoolingProcess,SecurityTrailer,
        SecurityTrailerWaitForReady,Available,AttachedToUser,
        DetachPending,OffLine),

TapeDriveStatus = (Available,AttachedToUser,OffLine,DetachPending),

ActivityStatus = (NotSpooling,Drained,Started,Draining),

Spoolid = structure of(
        Process = ProcessName,
        File = T"I:integer(0 <= I & I <= 999)),

ODRequestStatus = (Processing,WaitingForDevice),
OutputDeviceRequest = structure of (
          Process = ProcessName,
          RequestedClasses = set of Class,
          AttachedDevice = DeviceAddress,
          State = ODRequestStatus),

TapeDriveEntry = structure of (
          Raddr = DeviceAddress,
          State = TapeDriveStatus,
          AttachedProcess = ProcessName),

ResponseSlot = structure of (
          Respondent = ProcessName,
          Text = String,
          State = ResponseStatus),

PendingRequest = structure of (
          MsgId = MessageId,
          Kind = RequestCategory,
          Command = CommandName,
          Responses = set of ResponseSlot),

ReaderEntry = structure of (
          Raddr = DeviceAddress,
          State = ActivityStatus,
          CyclePosition = InputDeviceStatus,
          AttachedProcess = ProcessName,
          ClassesServedCurrently = set of Class,
          ClassesServedNextCycle = set of Class,
          ChannelStatusWord = HardwareStatus,
          LineBuffer = String),

PrinterEntry = structure of (
          Raddr = DeviceAddress,
          State = ActivityStatus,
          CyclePosition = OutputDeviceStatus,
          AttachedProcess = ProcessName,
          ClassesServedCurrently = set of Class,
          ClassesServedNextCycle = set of Class,
          RelinquishDeviceRequestState = RelDevRequestStatus,
          ChannelStatusWord = HardwareStatus),

PunchEntry = structure of (
          Raddr = DeviceAddress,
          State = ActivityStatus,
          CyclePosition = OutputDeviceStatus,
          AttachedProcess = ProcessName,
          ClassesServedCurrently = set of Class,
          ClassesServedNextCycle = set of Class,
          RelinquishDeviceRequestState = RelDevRequestStatus,
          ChannelStatusWord = HardwareStatus),
NkcpEntry = structure of (
        Process = ProcessName,
        UsableReaders = set of DeviceAddress,
        UsablePrinters = set of DeviceAddress,
        UsablePunches = set of DeviceAddress,
        UsableTapeDrives = set of DeviceAddress)

type
MessageLabel = (AddedNkcp,CannotAddNkcp,UserIdMapped,
        AddNkcp,DeleteNkcp,AttachDevice),

DeviceTypes,
KernelFunction,

Cat0 = T"(QUERY←UR,QUERY←ALL,QUERY←RADDR,QUERY←TAPES),

Cat1 = T"(QUERY←READER←SPOOLID,QUERY←PRINTER←SPOOLID,
        QUERY←PUNCH←SPOOLID,CHANGE←SYSTEM←SPOOLID),

Cat2 = T"(QUERY←FILES←ALL,QUERY←READER←ALL,QUERY←PRINTER←ALL,
        QUERY←PUNCH←ALL,QUERY←HOLD,CHANGE←SYSTEM←CLASS←ALL),

Cat3 = T"(QUERY←FILES←USERID,QUERY←READER←USERID,QUERY←PRINTER←USERID,
        QUERY←PUNCH←USERID,CHANGE←USERID,FREE,HOLD,ORDER←USERID,
        PURGE←USERID),

Cat4 = T"(BACKSPAC,FLUSH,REPEAT),

Cat6 = T"(ORDER←SYSTEM,PURGE←SYSTEM)

constant
MsgName(String):MessageLabel,
DeviceType(DeviceAddress):DeviceTypes,
Raddr(String):DeviceAddress,
NewMsgId:MessageId,
SendMessage(ProcessName):KernelFunction,
DefinedElsewhere:boolean = true,
Nkcps(String):set of ProcessName,
OpCmd(String):CommandName,
OpProcess,URProcess,AuthProcess,AcntProcess,UpdaterProcess:
        ProcessName,
TrustedProcesses = S"(OpProcess,URProcess,AuthProcess,
        AcntProcess,UpdaterProcess)
```

```
variable
ShuttingDown: boolean,
Readers: set of ReaderEntry,
Printers: set of PrinterEntry,
Punches: set of PunchEntry,
TapeDrives: set of TapeDriveEntry,
PrinterSpoolRequests: set of OutputDeviceRequest,
PunchSpoolRequests: set of OutputDeviceRequests,
CurrentNkcps: set of NkcpEntry,
PendingRequests: set of PendingRequest
```

```
initial
PendingRequests = Empty
&
CurrentNkcps = Empty
&
PrinterSpoolRequests = Empty
&
~ShuttingDown
&
A"R:ReaderEntry(R<:Readers ->
        (R.State = Drained
         &
         R.CyclePosition = Available
         &
         R.AttaxhedProcess = URProcess
         &
         R.LineBuffer = nil
         &
         R.ClassesServedCurrently = Empty
         &
         R.ClassesServedNextCycle = Empty))
&
A"P:PrinterEntry(P<Printers ->
        (P.State = Drained
         &
         P.CyclePosition = URProcess
         &
         P.AttachedProcess = URProcess
         &
         P.ClassesServedCurrently = Empty
         &
         P.ClassesServedNextCycle = Empty
         &
         P.RelinquishDeviceRequestState = NoNeed))
&
A"P:PunchEntry(P<:Punches ->
        (P.State = Drained
         &
         P.CyclePosition = URProcess
         &
         P.AttachedProcess = URProcess
         &
         P.ClassesServedCurrently = Empty
         &
         P.ClassesServedNextCycle = Empty
         &
         P.RelinquishDeviceRequestState = NoNeed))
&
A"T:TapeDriveEntry(T<:TapeDrives ->
        (T.State = Available
         &
         T.AttachedProcess = URProcess))
```

```
invariant
A"R1,R2:ReaderEntry(R1<:Readers & R2<:Readers ->
        (R1.Raddr = R2.Raddr -> R1 = R2))
&
A"Pr1,Pr2:PrinterEntry(Pr1<:Printers & Pr2<:Printers ->
        (Pr1.Raddr = Pr2.Raddr -> Pr1 = Pr2))
&
A"Pu1,Pu2:PunchEntry(Pu1<:Punches & Pu2<:Punches ->
        (Pu1.Raddr = Pu2.Raddr -> Pu1 = Pu2))
&
A"T1,T2:TapeDriveEntry(T1<:TapeDrives & T2<:TapeDrives ->
        (T1.Raddr = T2.Raddr -> T1 = T2))
&
A"R:ReaderEntry, Pr:PrinterEntry, Pu:PunchEntry, T:TapeDriveEntry
     (R<:Readers & Pr<:Printers & Pu<:Punches & T<:TapeDrives ->
        (R.Raddr ~= Pr.Raddr & R.Raddr ~= Pu.Raddr & R.Raddr ~= T.Raddr
         &
         Pr.Raddr ~= Pu.Raddr & Pr.Raddr ~= T.Raddr
         &
         Pu.Raddr ~= T.Raddr))
&
A"R:ReaderEntry(R<:Readers ->
        ((R.State = NotSpooling =>
             R.CyclePosition<:S"(AttachPending,DetachPending,
                                 AttachedToUser,OffLine)
          <> R.CyclePosition<:S"(SecurityHeader,
                                 SecurityHeaderWaitForReady,
                                 AttachPending,
                                 AttachedToSpoolingProcess,Available))
         &
         (R.CyclePosition = Available ->
                (R.ClassesServedCurrently = R.ClassesServedNextCycle
                 &
                 R.AttachedProcess = URProcess))))
&
A"P:PrinterEntry(P<:Printers ->
        ((P.State = NotSpooling =>
             P.CyclePosition<:S"(AttachPending,DetachPending,
                                  AttachedToUser,OffLine)
          <> P.CyclePosition<:S"(SecurityHeader,
                                 SecurityHeaderWaitForReady,
                                 SecurityTrailer,
                                 SecurityTrailerWaitForReady,
                                 AttachedToSpoolingProcess,Available))
         &
         (P.CyclePosition = Available ->
                (P.ClassesServedCurrently = P.ClassesServedNextCycle
                 &
                 P.Attach[illegible]ocess = URProcess))))
&
A"P:PunchEntry(P<:Punches ->
          ((P.State = NotSpooling =>
               P.CyclePosition<:S"(AttachPending,DetachPending,
                                   AttachedToUser,OffLine)
            <> P.CyclePosition<:S"(SecurityHeader,
                                   SecurityHeaderWaitForReady,
                                   SecurityTrailer,
                                   SecurityTrailerWaitForReady,
                                   AttachedToSpoolingProcess,Available))
          &
          (P.CyclePosition = Available ->
                (P.ClassesServedCurrently = P.ClassesServedNextCycle
                 &
                 P.AttachedProcess = URProcess))))
&
A"DR:OutputDeviceRequest((DR<:PrinterSpoolRequests
                                  |
                                  DR<:PunchSpoolRequests) ->
          (E"N:NkcpEntry(N<:CurrentNkcps &
             N.Process = DR.Process)))
&
A"N1,N2:NkcpEntry(N1<:CurrentNkcps & N2<:CurrentNkcps ->
          (N1.Process = N2.Process -> N1 = N2))
&
A"N:NkcpEntry(N<:CurrentNkcps ->
     (A"D:DeviceAddress
       ((D<:N.UsableReaders =>
                 E"R:ReaderEntry(R<:Readers &
                    D = R.Raddr)
         <> D<:N.UsablePrinters =>
                 E"Pr:PrinterEntry(Pr<:Printers &
                    D = Pr.Raddr)
         <> D<:N.UsablePunches =>
                 E"Pu:PunchEntry(Pu<:Punches &
                    D = Pu.Raddr)
         <> D<:N.UsableTapeDrives =>
                 E"T:TapeDriveEntry(T<:TapeDrives &
                    D = T.Raddr)))))
&
A"P1,P2:PendingRequest(P1<:PendingRequests & P2<:PendingRequests ->
          (P1.MsgId = P2.MsgId -> P1 = P2))
&
A"P:PendingRequest(P<:PendingRequests ->
          (A"R1,R2:ResponseSlot(R1<:P.Responses & R2<:P.Responses ->
                (R1.Respondent = R2.Respondent -> R1 = R2))))
```

```
transform KernelCalled(K:KernelFunction)

effect true
```

```
transform RD(Raddr: DeviceAddress)

refcond E"R:ReaderEntry(R<:Readers &
            R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
        (R.State<:S"(NotSpooling,Drained)
         |
         R.CyclePosition<:S"(AttachPending,AttachedToSpoolingProcess,
                             AttachedToUser,DetachPending,OffLine) =>
               Error
         <> (R.CyclePosition = SecurityHeader =>
                     (R.ChannelStatusWord.UnitCheck =>
                          RD2b
                      <>  RD2a)
              <> R.CyclePosition = SecurityHeaderWaitForReady =>
                     (R.ChannelStatusWord.UnitCheck =>
                          RD2b
                      <>  RD2c)
              <> R.CyclePosition = Available =>
                     (ShuttingDown =>
                          KernelCalled(SendMessage(OpProcess))
                      <> RD1(Raddr)))))
```

```
transform PR(Raddr: DeviceAddress)

refcond E"P:PrinterEntry(P<:Printers &
           P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
        (P.State<:S"(NotSpooling,Drained)
         |
         P.CyclePosition<:S"(AttachedToSpoolingProcess,
                             AttachedToUser,DetachPending,OffLine) =>
                Error
         <> (P.CyclePosition = SecurityHeader =>
                (P.ChannelStatusWord.UnitCheck =>
                    PR3b
                 <> PR3a)
         <> P.CyclePosition = SecurityHeaderWaitForReady =>
                (P.ChannelStausWord.UnitCheck =>
                    PR3b
                 <> PR3c)
         <> P.CyclePosition = Available =>
                (PrinterSpoolRequests ~= Empty
                 &
                 P.State = Started
                 &
                 E"DR:OutputDeviceRequest(DR<:PrinterSpoolRequests ->
                  (DR.State = WaitingForDevice
                   &
                   P.ClassesServedCurrently && DR.RequestedClasses
                           ~= Empty ->
                    PR2)))
         <> P.CyclePosition = SecurityTrailer =>
                (P.ChannelStatusWord.UnitCheck =>
                    PR5b
                 <> PR5a)
         <> P.CyclePosition = SecurityTrailerWaitForReady =>
                (P.ChannelStatusWord.UnitCheck =>
                    PR5b
                 <> PR5c))))
```

```
transform PU(Raddr: DeviceAddress)

refcond E"P:PunchEntry(P<:Punches &
          P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
         (P.State<:S"(NotSpooling,Drained)
          |
          P.CyclePosition<:S"(AttachedToSpoolingProcess,
                              AttachedToUser,DetachPending,OffLine) =>
                 Error
          <> (P.CyclePosition = SecurityHeader =>
                    (P.CHannelStatusWord.UnitCheck =>
                         PU3b
                     <>  PU3a)
          <> P.CyclePosition = SecurityHeaderWaitForReady =>
                    (P.Chann lStausWord.UnitCheck =>
                         PU3b
                     <>  PU3c)
          <> P.CyclePosition = Available =>
                    (PunchSpoolRequests ~= Empty
                     &
                     P.State = Started
                     &
                     E"DR:OutputDeviceRequest(DR<:PunchSpoolRequests ->
                       (DR.State = WaitingForDevice
                        &
                        P.ClassesServedCurrently && DR.RequestedClasses
                                  ~= Empty ->
                         PU2)))
          <> P.CyclePosition = SecurityTrailer =>
                    (P.ChannelStatusWord.UnitCheck =>
                         PU5b
                     <>  PU5a)
          <> P.CyclePosition = SecurityTrailerWaitForReady =>
                    (P.ChannelStatusWord.UnitCheck =>
                         PU5b
                     <>  PU5c))))
```

```
transform MsgAuth(MsgId: MessageId,
                  Text: String,
                  Source: ProcessName)

refcond Source = AuthProcess

effect   (E"P:PendingRequest(P<:PendingRequests &
           P.MsgId = MsgId) =>
           (E"P:PendingRequest(P<:PendingRequests & P.MsgId = MsgId &
             (P.Kind = NeedNkcp =>
                  (MsgName(Text) = AddNkcp =>
                        RO3a(Raddr(Text))
                        &
                        N"PendingRequests = PendingRequests ~~ S"(P)
                   <> MsgName(Text) = CannotAddNkcp =>
                        RO3b(Raddr(Text))
                        &
                        N"PendingRequests = PendingRequests ~~ S"(P)
                   <> Error)
              <> P.Kind = MapUserId =>
                  (MsgName(Text) = UserIdMapped =>
                        (Nkcp(Text) = nil =>
                             KernelCalled(SendMessage(OpProcess))
                             &
                              N"PendingRequests = PendingRequests
                                           ~ S"(P)
                         <> KernelCalled(SendMessage(Nkcp(Text)))
                             &
                             N"PendingRequsts = PendingRequests
                                          ~~ S"(P) ||
                                          S"((NewMsgId,
                                              OpRequest,
                                              OpCmd(Text),
                                              S"((Nkcp(Text),
                                                  nil,
                                                  NoResponse)))))
                   <> Error)
               <> Error)))
         <> /* message is a request, not a response */
            (MsgName(Text) = AddNkcp =>
                        AUTH1
             <> MsgName(Text) = DeleteNkcp =>
                        AUTH2
             <> MsgName(Text) = AttachDevice =>
                  (DeviceType(Raddr(Text)) = Reader =>
                        AUTH3a(Raddr(Text))
                   <> DeviceType(Raddr(Text)) = Printer =>
                        AUTH3b(Raddr(Text))
                   <> DeviceType(Raddr(Text)) = Punch =>
                        AUTH3c(Raddr(Text))
                   <> DeviceType(Raddr(Text)) = TapeDrive =>
                        AUTH3d(Raddr(Text))
```

```
          <> Error)
      <> Error))
```

```
transform MsgOp(MsgId: MessageId,
                Text: String,
                Source: ProcessName)

refcond Source = AuthProcess

effect  (E"P:PendingRequest(P<:PendingRequests &
           P.MsgId = MsgId) =>
               Error
         <> (MsgName(Text)<:Cat0 =>
                           OP0(MsgName(Text))
            <> MsgName(Text)<:Cat1 =>
                           OP1(MsgName(Text))
            <> MsgName(Text)<:Cat2 =>
                           OP2(MsgName(Text))
            <> MsgName(Text)<:Cat3 =>
                           OP3(MsgName(Text))
            <> MsgName(Text)<:Cat4 =>
                 (DeviceType(Raddr(Text)) = Printer =>
                           OP4a
                  <> DeviceType(Raddr(Text)) = Punch =>
                           OP4b
                  <>       Error)
            <> MsgName(Text) = SPACE =>
                 (DeviceType(Raddr(Text)) = Printer =>
                           OP5
                  <>       Error)
            <> MsgName(Text)<:Cat6 =>
                           OP6(MsgName(Text))
            <> MsgName(Text) = LOCATE←RADDR =>
                 (DeviceType(Raddr(Text)) = Reader =>
                           OP7a(MsgName(Text))
                  <> DeviceType(Raddr(Text)) = Printer =>
                           OP7b
                  <> DeviceType(Raddr(Text)) = Punch =>
                           OP7c
                  <> DeviceType(Raddr(Text)) = TapeDrive =>
                           OP7d
                  <> Error)
            <> MsgName(Text) = SHUTDOWN =>
                           OP8
            <> MsgName(Text) = VARY←OFFLINE =>
                 (DeviceType(Raddr(Text)) = Reader =>
                           OP9a
                  <> DeviceType(Raddr(Text)) = Printer =>
                           OP9b
                  <> DeviceType(Raddr(Text)) = Punch =>
                           OP9c
                  <> DeviceType(Raddr(Text)) = TapeDrive =>
                           OP9d
                  <> Error)
            <> MsgName(Text) = VARY←ONLINE =>
                 (DeviceType(Raddr(Text)) = Reader =>
                OP10a
              <> DeviceType(Raddr(Text)) = Printer =>
                OP10b
              <> DeviceType(Raddr(Text)) = Punch =>
                OP10c
              <> DeviceType(Raddr(Text)) = TapeDrive =>
                OP10d
              <> Error)
      <> MsgName(Text) = ATTACH←RADDR =>
             (DeviceType(Raddr(Text)) = Reader =>
                OP11a
              <> DeviceType(Raddr(Text)) = Printer =>
                OP11b
              <> DeviceType(Raddr(Text)) = Punch =>
                OP11c
              <> DeviceType(Raddr(Text)) = TapeDrive =>
                OP11d
              <> Error)
      <> MsgName(Text) = DETACH←RADDR =>
             (DeviceType(Raddr(Text)) = Reader =>
                OP12a
              <> DeviceType(Raddr(Text)) = Printer =>
                OP12b
              <> DeviceType(Raddr(Text)) = Punch =>
                OP12c
              <> DeviceType(Raddr(Text)) = TapeDrive =>
                OP12d
              <> Error)
      <> MsgName(Text) = LOADBUF =>
             (DeviceType(Raddr(Text)) = Printer =>
                OP13
              <> Error)
      <> MsgName(Text) = DRAIN =>
             (DeviceType(Raddr(Text)) = Reader =>
                OP14a
              <> DeviceType(Raddr(Text)) = Printer =>
                OP14b
              <> DeviceType(Raddr(Text)) = Punch =>
                OP14c
              <> Error)
      <> MsgName(Text) = START =>
             (DeviceType(Raddr(Text)) = Reader =>
                OP15a
              <> DeviceType(Raddr(Text)) = Printer =>
                OP15b
              <> DeviceType(Raddr(Text)) = Punch =>
                OP15c
              <> Error)
      <> MsgName(Text) = TRANSFER =>
                OP16
      <> Error))
```

```
transform MsgNkcp(MsgId: MessageId,
                  Text: String,
                  Source: ProcessName)

refcond Source~<:TrustedProcesses

effect  (E"P:PendingRequest(P<:PendingRequests &
           P.MsgId = MsgId) =>
          (E"P:PendingRequest(P<:PendingRequests & P.MsgId = MsgId &
            (P.Kind = OpRequest =>
                     MsgName(Text) = ResponseToOpRequest =>
                             ProcessedResponse(P,MsgId,Source)
                      <> Error)
             <> P.Kind = RelinquishDevice =>
                  (MsgName(Text) = DetachDevice =>
                     (DeviceType(Raddr(Text)) = Reader =>
                             NKCP1a
                      <> DeviceTYpe(Raddr(Text)) = Printer =>
                             NKCP1b
                      <> DeviceType(Raddr(Text)) = Punch =>
                             NKCP1c
                      <> DeviceType(Raddr(Text)) = TapeDrive =>
                             NKCP1d
                     <> Error)
                  <> Error)
            <> Error))
        <> /* message is not a response, but a request */
          (MsgName(Text) = DetachSpoolDevice =>
                  (DeviceType(Raddr(Text)) = Reader =>
                           RD4
                   <> DeviceType(Raddr(Text)) = Printer =>
                           PR4
                   <> DeviceType(Raddr(Text)) = Punch =>
                           PU4
                   <> Error)
            <> MsgName(Text) = DetachDevice =>
                  (DeviceType(Raddr(Text)) = Reader =>
                           NKCP2a
                   <> DeviceType(Raddr(Text)) = Printer =>
                           NKCP2b
                   <> DeviceType(Raddr(text)) = Punch =>
                           NKCP2c
                   <> DeviceType(Raddr(Text)) = TapeDrive =>
                           NKCP2d
                   <> Error)
            <> MsgName(Text) = NeedSpoolingDevice =>
                  (DeviceType(Raddr(Text)) = Printer =>
                           PR1
                   <> DeviceType(Raddr(Text)) = Punch =>
                           PU1
                   <> Error)
            <> Error))
```

```
transform URDriver(InterruptType: ,
                   InterruptSubType: ,
                   Raddr: DeviceAddress,
                   MsgId: MessageId,
                   Text: String,
                   Source: ProcessName)

effect  (InterruptType = IOInterrupt =>
                   (DeviceType(Raddr) = Reader =>
                            RD(Raddr)
                    <> DeviceType(Raddr) = Printer =>
                            PR(Raddr)
                    <> DeviceType(Raddr) = Punch =>
                            PU(Raddr)
                    <> Error)
          <> InterruptType= ExternalInterrupt =>
                 (InterruptSubType = Message =>
                   ReceivedMessage
                   &
                   (Source = AuthProcess =>
                            MsgAuth(MsgId,Text,Source)
                    <> Source = OpProcess =>
                            MsgOp(MsgId,Text,Source)
                    <> Source~<:TrustedProcesses =>
                            MsgNkcp(MsgId,Text,Source)
                    <> Error)
                 <> Error)
         <> Error)
        &
        KernelCalled(ReceiveInterrupts)
        &
        KernelCalled(ReleaseCPU)
```

RD1: Unexpected IO interrupt signalling deck to be read

```
transform RD1(Raddr: DeviceAddress)

refcond E"R:ReaderEntry (R<:Readers &
                   R.Raddr = Raddr
                    &
                    R.State = Started
                    &
                    R.CyclePosition = Available)
                &
                ~ ShuttingDown

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
            N"Readers = Readers ~~ S"(R) ||
              S"((R.Raddr,
                  R.State,
        /*##*/    SecurityHeader,
                  R.AttachedProcess,
                  R.ClassesServedCurrently,
                  R.ClassesServedNextCycle,
                  R.ChannelStatusWord,
        /*##*/    nil))
            &
            KernelCalled(RequestIO(R.Raddr)))
```

RD2a: Expected IO interrupt signalling security header read

```
transform RD2a(Raddr: DeviceAddress,
               Process: ProcessName)

refcond E"R:ReaderEntry (R<:Readers &
                   R.Raddr = Raddr
                    &
                    R.State ~= NotSpooling
                    &
                    R.CyclePosition = SecurityHeader
                    &
                    R.ChannelStatusWord.UnitCheck = false
                    &
                    R.LineBuffer ~= nil)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                    (E"N:NkcpEntry(N<:CurrentNkcps &
                       N.Process = Process) =>
                    E"N:NkcpEntry(N<:CurrentNkcps & N.Process = Process &
            N"Readers = Readers ~~ S"(R) ||
              S"((R.Raddr,
        /*##*/    ((A"D:DeviceAddress(D<:N.UsableReaders -> D ~= R.Raddr)
                        |
                       ~GrantedAccess)
                    &
                    R.State = Draining =>
                       Drained
                     <> R.State),
        /*##*/    (E"D:DeviceAddress(D<:N.UsableReaders & D = R.Raddr)
                    &
                    GrantedAccess =>
                       AttachedToSpoolingProcess
                     <> Available),
        /*##*/    (E"D:DeviceAddress(D<:N.UsableReaders & D = R.Raddr)
                    &
                    GrantedAccess =>
                       N.Process
                     <> URProcess),
        /*##*/    (A"D:DeviceAddress(D<:N.UsableReaders -> D ~= R.Raddr)
                    |
                    ~GrantedAccess =>
                       R.ClassesServedNextCycle
                     <> R.ClassesServedCurrently),
                  R.ClassesServedNextCycle,
                  R.ChannelStatusWord,
                  R.LineBuffer))
            &
```

```
                          (E"D:DeviceAddress(D<:N.UsableReaders & D = R.Raddr) =>
                              KernelCalled(GrantAccess)
                              &
                              (GrantedAccess =>
                                     KernelCalled(SendMessage(N.Process))
                                   <> KernelCalled(SendMessage(OpProcess))
                                     &
                                     (R.State = Draining -> KernelCalled(
                                                  SendMessage(OpProcess))))
                           <> KernelCalled(SendMessage(OpProcess))
                              &
                              (R.State = Draining -> KernelCalled(
                                           SendMessage(OpProcess)))))
              <>
                  N"Readers = Readers ~~ S"(R) ||
                    S"((R.Raddr,
                        R.State,
              /*##*/    AttachPending,
              /*##*/    Process,
                        R.ClassesServedCurrently,
                        R.ClassesServedNextCycle,
                        R.ChannelStatusWord,
                        R.LineBuffer))
                  &
                          KernelCalled(SendMessage(AuthProcess))))
```

RD2b: IO Error when attempting security header read

```
transform RD2b(Raddr: DeviceAddress)

refcond E"R:ReaderEntry (R<:Readers &
                    R.Raddr = Raddr
                     &
                     R.State ~= NotSpooling
                     &
                     R.CyclePosition<:S"(SecurityHeader,
                                         SecurityHeaderWaitForReady)
                     &
                     R.ChannelStatusWord.UnitCheck = true)

effect   E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
              N"Readers = Readers ~~ S"(R) ||
                S"((R.Raddr,
                    R.State,
         /*##*/     SecurityHeaderWaitForReady,
                    R.AttachedProcess,
                    R.ClassesServedCurrently,
                    R.ClassesServedNextCycle,
                    R.ChannelStatusWord,
                    R.LineBuffer)))
         &
         KernelCalled(SendMessage(OpProcess))
```

RD2c: Reader error cleared by operator (IO interrupt signalling device ready)

```
transform RD2c(Raddr: DeviceAddress)

refcond E"R:ReaderEntry (R<:Readers &
                    R.Raddr = Raddr
                     &
                     R.State ~= NotSpooling
                     &
                     R.CyclePosition = SecurityHeaderWaitForReady
                     &
                     R.ChannelStatusWord.UnitCheck = false)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
            N"Readers = Readers ~~ S"(R) ||
              S"((R.Raddr,
                 R.State,
        /*##*/   SecurityHeader,
                 R.AttachedProcess,
                 R.ClassesServedCurrently,
                 R.ClassesServedNextCycle,
                 R.ChannelStatusWord,
        /*##*/   nil))
            &
            KernelCalled(RequestIO(R.Raddr)))
```

RD3a: AuthProcess message re Nkcp creation: added

```
transform RD3a(Raddr: DeviceAddress)

refcond E"R:ReaderEntry (R<:Readers &
                     R.Raddr = Raddr
                 &
                 E"N:NkcpEntry(N<:CurrentNkcps &
                     N.Process = R.AttachedProcess))

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                 (~(R.State ~= NotSpooling
                     &
                     R.CyclePosition = AttachPending)
                 => Error <>
                        E"N:NkcpEntry(N<:CurrentNkcps
                                        &
                                        N.Process = R.AttachedProcess
                                        &
             N"Readers = Readers ~~ S"(R) ||
                S"((R.Raddr,
    /*##*/          ((A"D:DeviceAddress(D<:N.UsableReaders -> D ~= R.Raddr)
                            |
                            ~GrantedAccess)
                     &
                     R.State = Draining =>
                            Drained
                       <> R.State),
    /*##*/          (E"D:DeviceAddress(D<:N.UsableReaders & D = R.Raddr)
                     &
                     GrantedAccess =>
                            AtttachedToSpoolingProcess
                       <> Available),
    /*##*/          (A"D:DeviceAddress(D<:N.UsableReaders -> D ~= R.Raddr)
                     |
                     ~GrantedAccess =>
                            URProcess
                       <> R.AttachedProcess),
    /*##*/          (A"D:DeviceAddress(D<:N.UsableReaders -> D ~= R.Raddr)
                     |
                     ~GrantedAccess =>
                            R.ClassesServedNextCycle
                       <> R.ClassesServedCurrently),
                    R.ClassesServedNextCycle,
                    R.ChannelStatusWord,
                    R.LineBuffer))
             &
```

```
                KernelCalled(SendMessage(N.Process))
         &
                (E"D:DeviceAddress(D<:N.UsableReaders &  D = R.Raddr) =>
                    KernelCalled(GrantAccess)
                    &
                    (~GrantedAccess ->
                            KernelCalled(SendMessage(OpProcess)))
                 <> KernelCalled(SendMessage(OpProcess))))))
```

RD3b: AuthProcess message re Nkcp creation: cannot add

```
transform RD3b(Raddr: DeviceAddress)

refcond E"R:ReaderEntry (R<:Readers &
                         R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                (~(R.State ~= NotSpooling
                   &
                   R.CyclePosition = AttachPending)
                => Error <>
            N"Readers = Readers ~~ S"(R) ||
              S"((R.Raddr,
        /*##*/      (R.State = Draining =>
                          Drained
                       <> R.State),
        /*##*/      Available,
        /*##*/      URProcess,
        /*##*/      R.ClassesServedNextCycle,
                    R.ClassesServedNextCycle,
                    R.ChannelStatusWord,
                    R.LineBuffer)))
        &
        KernelCalled(SendMessage(OpProcess)))
```

RD4: Process releases reader after spooling

```
transform RD4(Raddr: DeviceAddress,
              RequestingProcess: ProcessName)

refcond E"R:ReaderEntry (R<:Readers &
                         R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
              (~(R.State ~= NotSpooling
                   &
                   R.CyclePosition = AttachedtoSpoolingProcess
                   &
                   R.AttachedProcess = RequestingProcess)
              => Error <>
           N"Readers = Readers ~~ S"(R) ||
             S"((R.Raddr,
        /*##*/    (DeviceReleased & R.State = Draining =>
                        Drained
                     <> R.State),
        /*##*/    (DeviceReleased =>
                        Available
                     <> R.CyclePosition),
        /*##*/    (DeviceReleased =>
                        URProcess
                     <> R.AttachedProcess),
                  R.ClassesServedCurrently,
                  R.ClassesServedNextCycle,
                  R.ChannelStatusWord,
                  R.LineBuffer))
           &
                  KernelCalled(ReleaseDevice)
           &
                  (R.State = Draining | ~Device Released =>
                        KernelCalled(SendMessage(OpProcess))
                     <> KernelCalled(SendMessage(R.AttachedProcess)))))
```

OP14a: Drain (Reader)

```
transform OP14a(Raddr: DeviceAddress)

refcond E"R:ReaderEntry (R<:Readers &
                        R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                     (R.State = NotSpooling
                     => Error <>
                 N"Readers = Readers ~~ S"(R) ||
                   S"((R.Raddr,
        /*##*/       (R.CyclePosition = Available =>
                          Drained
                        <> Draining),
                      R.CyclePosition,
                      R.AttachedProcess,
                      R.ClassesServedCurrently,
                      R.ClassesServedNextCycle,
                      R.ChannelStatusWord,
                      R.LineBuffer))
                 &
                        (R.CyclePosition = Available ->
                                 KernelCalled(SendMessage(OpProcess)))))
```

OP15a: Start (Reader)

```
transform OP15a(Raddr: DeviceAddress,
                NewClasses: set of Class)

refcond E"R:ReaderEntry (R<:Readers &
                         R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                (R.State = NotSpooling
                => Error <>
             N"Readers = Readers ~~ S"(R) ||
               S"((R.Raddr,
        /*####*/     (NewClasses ~= Empty
                       |
                      R.ClassesServedCurrently ~= Empty =>
                         Started
                       <> R.State),
                    R.CyclePosition,
                    R.AttachedProcess,
        /*####*/     (NewClasses ~= Empty
                       &
                      R.CyclePosition = Available =>
                         NewClasses
                       <> R.ClassesServedCurrently),
        /*####*/     (NewClasses ~= Empty =>
                         NewClasses
                       <> R.ClassesServedNextCycle),
                    R.ChannelStatusWord,
                    R.LineBuffer))
        &
        KernelCalled(SendMessage(OpProcess))))
```

PR1: Process request for output spooling device assignment

```
transform PR1(Process: ProcessName
              RequestedClasses: set of Class)

effect  (A"N:NkcpEntry (N<:CurrentNkcps ->
                   N.Process ~= Process)
              |
              RequestedClasses = Empty
              => Error <>
        N"PrinterSpoolRequests =
          (~ShuttingDown =>
              PrinterSpoolRequests ||
                S"((Process,
                   RequestedClasses,
                   nil,
                   WaitingForDevice))
           <> PrinterSpoolRequests))
```

PR2: Printer Assignment (for spooling)

```
transform PR2(DR: OutputDeviceRequest,
              P: PrinterEntry)

refcond DR<:PrinterSpoolRequests
              &
              P<:Printers
              &
              P.State = Started
              &
              P.CyclePosition = Available
              &
              DR.State = WaitingForDevice
              &
              ClassesMatch(P.ClassesServedCurrently,
                     DR.RequestedClasses)
              &
              E"N:NkcpEntry(N<:CurrentNkcps &
                 N.Process = DR.Process)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                (E"N:NkcpEntry(N<:CurrentNkcps & N.Process = DR.Process &
          N"Printers = Printers ~~ S"(P) ||
            S"((P.Raddr,
               P.State,
     /*##*/     (E"D:DeviceAddress(D<:N.UsablePrinters & D = P.Raddr) =>
                    SecurityHeader
                  <> P.CyclePosition),
     /*##*/     (E"D:DeviceAddress(D<:N.UsablePrinters & D = P.Raddr) =>
                    DR.Process
                  <> P.AttachedProcess),
               P.ClassesServedCurrently,
               P.ClassesServedNextCycle,
               P.RelinquishDeviceRequestState,
               P.ChannelStatusWord))
          &
          N"PrinterSpoolRequests = PrinterSpoolRequests ~~ S"(DR) ||
            S"((DR.Process,
               DR.RequestedClasses,
     /*##*/     (E"D:DeviceAddress(D<:N.UsablePrinters & D = P.Raddr) =>
                    P.Raddr
                  <> DR.AttachedDevice)
     /*##*/     (E"D:DeviceAddress(D<:N.UsablePrinters & D = P.Raddr) =>
                    Processing
                  <> DR.State))
          &
                KernelCalled(RequestIO(P.Raddr))))
```

PR3a: Interrupt indicating end of security header output on printer

```
transform PR3a(Raddr: DeviceAddress)

refcond E"P:PrinterEntry (P<:Printers &
                          P.Raddr = Raddr
                          &
                          P.State ~= NotSpooling
                          &
                          P.CyclePosition = SecurityHeader
                          &
                          P.ChannelStatusWord.UnitCheck = false)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
             N"Printers = Printers ~~ S"(P) ||
               S"((P.Raddr,
                   P.State,
        /*##*/     (GrantedAccess =>
                        AttachedToSpoolingProcess
                     <> SecurityTrailer),
                   P.AttachedProcess,
                   P.ClassesServedCurrently,
                   P.ClassesServedNextCycle,
        /*##*/     (GrantedAccess
                    &
                    P.RelinquishDeviceRequestState = ShouldSend =>
                        Sent
                     <> P.RelinquishDeviceRequestState),
                   P.ChannelStatusWord))
             &
                    (GrantedAccess =>
                        KernelCalled(SendMessage(P.AttachedProcess))
                        &
                        (P.RelinquishDeviceRequestState = ShouldSend ->
                              KernelCalled(SendMessage(P.AttachedProcess)))
                     <> KernelCalled(RequestIO(P.Raddr))))
```

PR3b: Interrupt, IO error on attempt to output security header

```
transform PR3b(Raddr: DeviceAddress)

refcond E"P:PrinterEntry (P<:Printers &
                          P.Raddr = Raddr
                          &
                          P.State ~= NotSpooling
                          &
                          P.CyclePosition<:S"(SecurityHeader,
                                              SecurityHeaderWaitForReady)
                          &
                          P.ChannelStatusWord.UnitCheck = true)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
             N"Printers = Printers ~~ S"(P) ||
               S"((P.Raddr,
                   P.State,
        /*##*/     SecurityHeaderWaitForReady,
                   P.AttachedProcess,
                   P.ClassesServedCurrently,
                   P.ClassesServedNextCycle,
                   P.RelinquishDeviceRequestState,
                   P.ChannelStatusWord))
             &
                   KernelCalled(SendMessage(OpProcess)))
```

PR3c: Interrupt indicating OK to retry security header output

```
transform PR3c(Raddr: DeviceAddress)

refcond E"P:PrinterEntry (P<:Printers &
                    P.Raddr = Raddr
                    &
                    P.State ~= NotSpooling
                    &
                    P.CyclePosition = SecurityHeaderWaitForReady
                    &
                    P.ChannelStatusWord.UnitCheck = false)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
             N"Printers = Printers ~~ S"(P) ||
               S"((P.Raddr,
                   P.State,
        /*##*/     SecurityHeader,
                   P.AttachedProcess,
                   P.ClassesServedCurrently,
                   P.ClassesServedNextCycle,
                   P.RelinquishDeviceRequestState,
                   P.ChannelStatusWord))
             &
                    KernelCalled(RequestIO(P.Raddr)))
```

PR4: Process message, release output spooling device

```
transform PRPU4(Raddr: DeviceAddress,
                Process: ProcessName)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
               (~(E"P:PrinterEntry (P<:Printers &
                    P.Raddr = Raddr
                    &
                    P.State ~= NotSpooling
                    &
                    P.CyclePosition = AttachedToSpoolingProcess
                    &
                    P.AttachedProcess = Process
               &
               E"DR:OutputDeviceRequest(DR<:PrinterSpoolRequests &
                    DR.Process = P.AttachedProcess
                    &
                    DR.State = Processing
                    &
                    DR.AttachedDevice = P.Raddr)))
               => Error <>
          N"Printers = Printers ~~ S"(P) ||
            S"((P.Raddr,
               P.State,
      /*###*/  SecurityTrailer,
      /*###*/  URProcess,
               P.ClassesServedCurrently,
               P.ClassesServedNextCycle,
               P.RelinquishDeviceRequestState,
               P.ChannelStatusWord))
        &
                 E"DR:OutputDeviceRequest(DR<:PrinterSpoolRequests &
                      DR.Process = P.AttachedProcess
        &
                      DR.State = Processing
        &
                      DR.AttachedDevice = P.Raddr
        &
                      N"PrinterSpoolRequests =
                            PrinterSpoolRequests ~~S"(DR))
        &
                 KernelCalled(RequestIO(P.Raddr))))
```

PRSa: Interrupt indicating successful completion of security trailer

```
transform PRSa(Raddr: DeviceAddress)

refcond E"P:PrinterEntry(P<:Printers &
                  P.Raddr = Raddr
                  &
                  P.State ~= NotSpooling
                  &
                  P.CyclePositicn = SecurityTrailer
                  &
                  P.ChannelStatusWord.UnitCheck = false)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
            N"Printers = Printers ~~ S"(P) ||
              S"((P.Raddr,
        /*##*/     (P.State = Draining =>
                         Drained
                      <> P.State),
        /*##*/    Available,
                  P.AttachedProcess,
        /*##*/    P.ClassesServedNextCycle,
                  P.ClassesServedNextCycle,
        /*##*/    NoNeed,
                  P.ChannelStatusWord)))
```

PR5b: Interrupt. IO error on security trailer output

```
transform PR5b(Raddr: DeviceAddress)

refcond E"P:PrinterEntry (P<:Printers &
                        P.Raddr = Raddr
                        &
                        P.State ~= NotSpooling
                        &
                        P.CyclePosition<:S"(SecurityTrailer,
                                            SecurityTrailerWaitForReady)
                        &
                        P.ChannelStatusWord.UnitCheck = true)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
              N"Printers = Printers ~~ S"(P) ||
                S"((P.Raddr,
                      P.State,
        /*##*/        SecurityTrailerWaitForReady,
                      P.AttachedProcess,
                      P.ClassesServedCurrently,
                      P.ClassesServedNextCycle,
                      P.RelinquishDeviceRequestState,
                      P.ChannelStatusWord))
              &
                        KernelCalled(SendMessage(OpProcess)))
```

PRSc: Interrupt, OK to retry security trailer output

```
transform PRSc(Raddr: DeviceAddress)

refcond E"P:PrinterEntry (P<:Printers &
                          P.Raddr = Raddr
                          &
                          P.State ~= NotSpooling
                          &
                          P.CyclePosition = SecurityTrailerWaitForReady
                          &
                          P.ChannelStatusWord.UnitCheck = false)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
             N"Printers = Printers ~~ S"(P) ||
               S"((P.Raddr,
                   P.State,
        /*##*/     SecurityTrailer,
                   P.AttachedProcess,
                   P.ClassesServedCurrently,
                   P.ClassesServedNextCycle,
                   P.RelinquishDeviceRequestState,
                   P.ChannelStatusWord))
             &
                      KernelCalled(RequestIO(P.Raddr)))
```

PU1: Process request for output spooling device assignment

```
transform PU1(Process: ProcessName
              RequestedClasses: set of Class)

effect  (A"N:NkcpEntry (N<:CurrentNkcps ->
                N.Process ~= Process)
             |
             RequestedClasses = Empty
             => Error <>
        N"PunchSpoolRequests =
          (~ShuttingDown =>
             PunchSpoolRequests ||
               S"((Process,
                   RequestedClasses,
                   nil,
                   WaitingForDevice))
           <> PunchSpoolRequests))
```

PU2: Punch Assignment (for spooling)

```
transform PU2(DR: OutputDeviceRequest,
              P: PunchEntry)

refcond DR<:PunchSpoolRequests
              &
              P<:Punches
              &
              P.State = Started
              &
              P.CyclePosition = Available
              &
              DR.State = WaitingForDevice
              &
              ClassesMatch(P.ClassesServedCurrently,
                     DR.RequestedClasses)
              &
              E"N:NkcpEntry(N<:CurrentNkcps &
                  N.Process = DR.Process)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                  (E"N:NkcpEntry(N<:CurrentNkcps & N.Process = DR.Process &
           N"Punches = Punches ~~ S"(P) ||
             S"((P.Raddr,
                 P.State,
     /*##*/      (E"D:DeviceAddress(D<:N.UsablePunches & D = P.Raddr) =>
                        SecurityHeader
                     <> P.CyclePosition),
     /*##*/      (E"D:DeviceAddress(D<:N.UsablePunches & D = P.Raddr) =>
                        DR.Process
                     <> P.AttachedProcess),
                 P.ClassesServedCurrently,
                 P.ClassesServedNextCycle,
                 P.RelinquishDeviceRequestState,
                 P.ChannelStatusWord))
           &
           N"PunchSpoolRequests = PunchSpoolRequests ~~ S"(DR) ||
             S"((DR.Process,
                 DR.RequestedClasses,
     /*##*/      (E"D:DeviceAddress(D<:N.UsablePunches & D = P.Raddr) =>
                        P.Raddr
                     <> DR.AttachedDevice)
     /*##*/     (E"D:DeviceAddress(D<:N.UsablePunches & D = P.Raddr) =>
                        Processing
                     <> DR.State)))
           &
                 KernelCalled(RequestIO(P.Raddr)))))
```

PU3a: Interrupt indicating end of security header output on punch

```
transform PU3a(Raddr: DeviceAddress)

refcond E"P:PunchEntry (P<:Punches &
                        P.Raddr = Raddr
                        &
                        P.State ~= NotSpooling
                        &
                        P.CyclePosition = SecurityHeader
                        &
                        P.ChannelStatusWord.UnitCheck = false)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
            N"Punches = Punches ~~ S"(P) ||
              S"((P.Raddr,
                  P.State,
        /*##*/    (GrantedAccess =>
                        AttachedToSpoolingProcess
                     <> SecurityTrailer),
                  P.AttachedProcess,
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
        /*##*/    (GrantedAccess
                    &
                    P.RelinquishDeviceRequestState = ShouldSend =>
                        Sent
                     <> P.RelinquishDeviceRequestState),
                  P.ChannelStatusWord))
            &
                   (GrantedAccess =>
                        KernelCalled(SendMessage(P.AttachedProcess))
                        &
                        (P.RelinquishDeviceRequestState = ShouldSend ->
                               KernelCalled(SendMessage(P.AttachedProcess)))
                     <> KernelCalled(RequestIO(P.Raddr))))
```

PU3b: Interrupt, IO error on attempt to output security header

```
transform PU3b(Raddr: DeviceAddress)

refcond E"P:PunchEntry (P<:Punches &
                        P.Raddr = Raddr
                        &
                        P.State ~= NotSpooling
                        &
                        P.CyclePosition<:S"(SecurityHeader,
                                            SecurityHeaderWaitForReady)
                        &
                        P.ChannelStatusWord.UnitCheck = true)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
               N"Punches = Punches ~~ S"(P) ||
                 S"((P.Raddr,
                     P.State,
               */    SecurityHeaderWaitForReady,
                     P.AttachedProcess,
                     P.ClassesServedCurrently,
                     P.ClassesServedNextCycle,
                     P.RelinquishDeviceRequestState,
                     P.ChannelStatusWord))
               &
                        KernelCalled(SendMessage(OpProcess)))
```

PU3c: Interrupt indicating OK to retry security header output

```
transform PU3c(Raddr: DeviceAddress)

refcond E"P:PunchEntry (P<:Punches &
                       P.Raddr = Raddr
                       &
                       P.State ~= NotSpooling
                       &
                       P.CyclePosition = SecurityHeaderWaitForReady
                       &
                       P.ChannelStatusWord.UnitCheck = false)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
             N"Punches = Punches ~~ S"(P) ||
               S"((P.Raddr,
                   P.State,
        /*##*/     SecurityHeader,
                   P.AttachedProcess,
                   P.ClassesServedCurrently,
                   P.ClassesServedNextCycle,
                   P.RelinquishDeviceRequestState,
                   P.ChannelStatusWord))
             &
                       KernelCalled(RequestIO(P.Raddr)))
```

PU4: Process message, release output spooling device

```
transform PU4(Raddr: DeviceAddress,
                   Process: ProcessName)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                  (~(E"P:PunchEntry (P<:Punches &
                      P.Raddr = Raddr
                      &
                      P.State ~= NotSpooling
                      &
                      P.CyclePosition = AttachedToSpoolingProcess
                      &
                      P.AttachedProcess = Process
                  &
                  E"DR:OutputDeviceRequest(DR<:PunchSpoolRequests &
                      DR.Process = P.AttachedProcess
                      &
                      DR.State = Processing
                      &
                      DR.AttachedDevice = P.Raddr)))
                  => Error <>
            N"Punches = Punches ~~ S"(P) ||
              S"((P.Raddr,
                  P.State,
     /*##*/       SecurityTrailer,
     /*##*/       URProcess,
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
                  P.RelinquishDeviceRequestState,
                  P.ChannelStatusWord))
            &
                  E"DR:OutputDeviceRequest(DR<:PunchSpoolRequests &
                        DR.Process = P.AttachedProcess
            &
                        DR.State = Processing
            &
                        DR.AttachedDevice = P.Raddr
            &
                        N"PunchSpoolRequests =
                               PunchSpoolRequests ~~S"(DR))
            &
                  KernelCalled(RequestIO(P.Raddr))))
```

PU5a: Interrupt indicating successful completion of security trailer

```
transform PU5a(Raddr: DeviceAddress)

refcond E"P:PunchEntry(P<:Punches &
                P.Raddr = Raddr
                &
                P.State ~= NotSpooling
                &
                P.CyclePosition = SecurityTrailer
                &
                P.ChannelStatusWord.UnitCheck = false)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
            N"Punches = Punches ~~ S"(P) ||
              S"((P.Raddr,
        /*##*/      (P.State = Draining =>
                          Drained
                       <> P.State),
        /*##*/     Available,
                   P.AttachedProcess,
        /*##*/     P.ClassesServedNextCycle,
                   P.ClassesServedNextCycle,
        /*##*/     NoNeed,
                   P.ChannelStatusWord)))
```

PU5b: Interrupt, IO error on security trailer output

```
transform PU5b(Raddr: DeviceAddress)

refcond E"P:PunchEntry (P<:Punches &
                     P.Raddr = Raddr
                     &
                     P.State ~= NotSpooling
                     &
                     P.CyclePosition<:S"(SecurityTrailer,
                                         SecurityTrailerWaitForReady)
                     &
                     P.ChannelStatusWord.UnitCheck = true)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
              N"Punches = Punches ~~ S"(P) ||
                S"((P.Raddr,
                    P.State,
        /*##*/      SecurityTrailerWaitForReady,
                    P.AttachedProcess,
                    P.ClassesServedCurrently,
                    P.ClassesServedNextCycle,
                    P.RelinquishDeviceRequestState,
                    P.ChannelStatusWord))
              &
                     KernelCalled(SendMessage(OpProcess)))
```

PU5c: Interrupt, OK to retry security trailer output

```
transform PU5c(Raddr: DeviceAddress)

refcond E"P:PunchEntry (P<:Punches &
                        P.Raddr = Raddr
                        &
                        P.State ~= NotSpooling
                        &
                        P.CyclePosition = SecurityTrailerWaitForReady
                        &
                        P.ChannelStatusWord.UnitCheck = false)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
             N"Punches = Punches ~~ S"(P) ||
               S"((P.Raddr,
                   P.State,
        /*##*/     SecurityTrailer,
                   P.AttachedProcess,
                   P.ClassesServedCurrently,
                   P.ClassesServedNextCycle,
                   P.RelinquishDeviceRequestState,
                   P.ChannelStatusWord))
             &
                    KernelCalled(RequestIO(P.Raddr)))
```

OP14b: Drain (Printer)

```
transform OP14b(Raddr: DeviceAddress)

refcond E"P:PrinterEntry (P<:Printers &
                        P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
              (P.State = NotSpooling
              => Error <>
          N"Printers = Printers ~~ S"(P) ||
            S"((P.Raddr,
     /*##*/      (P.CyclePosition = Available =>
                      Drained
                   <> Draining),
                 P.CyclePosition,
                 P.AttachedProcess,
                 P.ClassesServedCurrently,
                 P.ClassesServedNextCycle,
     /*##*/      (P.CyclePosition = AttachedToSpoolingProcess =>
                      Sent
                   <> (P.CyclePosition = Available =>
                             P.RelinquishDeviceRequestState
                         <> ShouldSend)),
                 P.ChannelStatusWord))
          &
                    (P.CyclePosition = Available ->
                        KernelCalled(SendMessage(OpProcess)))
          &
                    (P.CyclePosition = AttachedToSpoolingProcess
                     &
                     P.RelinquishDeviceRequestState ~= Sent ->
                        KernelCalled(SendMessage(P.AttachedProcess))))
```

OP14c: Drain (Punch)

```
transform OP14c(Raddr: DeviceAddress)

refcond E"P:PunchEntry (P<:Punches &
                        P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                  (P.State = NotSpooling
                  => Error <>
            N"Punches = Punches ~~ S"(P) ||
              S"((P.Raddr,
        /*##*/      (P.CyclePosition = Available =>
                          Drained
                       <> Draining),
                    P.CyclePosition,
                    P.AttachedProcess,
                    P.ClassesServedCurrently,
                    P.ClassesServedNextCycle,
        /*##*/      (P.CyclePosition = AttachedToSpoolingProcess =>
                          Sent
                       <> (P.CyclePosition = Available =>
                                 P.RelinquishDeviceRequestState
                              <> ShouldSend)),
                    P.ChannelStatusWord))
            &
                      (P.CyclePosition = Available ->
                          KernelCalled(SendMessage(OpProcess)))
            &
                      (P.CyclePosition = AttachedToSpoolingProcess
                       &
                       P.RelinquishDeviceRequestState ~= Sent ->
                          KernelCalled(SendMessage(P.AttachedProcess)))))
```

OP15b: Start (Printer)

```
transform OP15b(Raddr: DeviceAddress,
                NewClasses: set of Class)

refcond E"P:PrinterEntry (P<:Printers &
                    P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                (P.State = NotSpooling
                => Error <>
             N"Printers = Printers ~~ S"(P) ||
               S"((P.Raddr,
     /*##*/       (NewClasses ~= Empty
                    |
                   P.ClassesServedCurrently ~= Empty =>
                       Started
                     <> P.State),
                  P.CyclePosition,
                  P.AttachedProcess,
     /*##*/       (P.CyclePosition = Available
                    &
                   NewClasses ~= Empty =>
                       NewClasses
                     <> P.ClassesServedCurrently),
     /*##*/       (NewClasses ~= Empty =>
                       NewClasses
                     <> P.ClassesServedNextCycle),
     /*##*/       (NewClasses ~= Empty =>
                       (P.CyclePosition = AttachedToSpoolingProcess =>
                              Sent
                          <> (P.CyclePosition = Available =>
                                     P.RelinquishDeviceRequestState
                                  <> ShouldSend))
                     <> P.RelinquishDeviceRequestState),
                  P.ChannelStatusWord))
          &
                   KernelCalled(SendMessage(OpProcess))
          &
                    (NewClasses ~= Empty
                     &
                     P.CyclePosition = AttachedToSpoolingProcess
                     &
                     P.RelinquishDeviceRequestState ~= Sent ->
                        KernelCalled(SendMessage(P.AttachedProcess)))))
```

OP15c: Start (Punch)

```
transform OP15c(Raddr: DeviceAddress,
                NewClasses: set of Class)

refcond E"P:PunchEntry (P<:Punches &
                    P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
              (P.State = NotSpooling
              => Error <>
          N"Punches = Punches ~~ S"(P) ||
            S"((P.Raddr,
       /*##*/     (NewClasses ~= Empty
                   |
                   P.ClassesServedCurrently ~= Empty =>
                      Started
                    <> P.State),
                 P.CyclePosition,
                 P.AttachedProcess,
       /*##*/    (P.CyclePosition = Available
                   &
                   NewClasses ~= Empty =>
                      NewClasses
                    <> P.ClassesServedCurrently),
       /*##*/    (NewClasses ~= Empty =>
                      NewClasses
                    <> P.ClassesServedNextCycle),
       /*##*/    (NewClasses ~= Empty =>
                      (P.CyclePosition = AttachedToSpoolingProcess =>
                             Sent
                          <> (P.CyclePosition = Available =>
                                    P.RelinquishDeviceRequestState
                               <> ShouldSend))
                    <> P.RelinquishDeviceRequestState),
                 P.ChannelStatusWord))
          &
                KernelCalled(SendMessage(OpProcess))
          &
                 (NewClasses ~= Empty
                  &
                  P.CyclePosition = AttachedToSpoolingProcess
                  &
                  P.RelinquishUeviceRequestState ~= Sent ->
                     KernelCalled(SendMessage(P.AttachedProcess)))))
```

```
OP0:            Miscellaneous commands

        commands:
                QUERY←UR
                QUERY←ALL
                QUERY←RADOR
                QUERY←TAPES

transform OP0(Command: CommandName)

refcond Command<:Cat0

effect  KernelCalled(SendMessage(OpProcess))
```

```
OP1:            Single message sent
                No maps
                Single response expected
                No device state information modifications

        commands:
                QUERY←READER←SPOOLID
                QUERY←PRINTER←SPOOLID
                QUERY←PUNCH←SPOOLID
                CHANGE←SYSTEM←SPOOLID

transform OP1(Command: CommandName)

refcond Command<:Cat1

effect  N"PendingRequests = PendingRequests ||
              S"((NewMsgId,
                  OpRequest,
                  Command,
                  S"((Destination(Command),
                      nil,
                      NoResponse))))
        &
        KernelCalled(SendMessage(Destination(Command)))
```

```
OP2:            Multiple messages sent
                No maps
                Responses expected
                No device state information modifications

        commands:
                QUERY←FILES←ALL
                QUERY←READER←ALL
                QUERY←PRINTER←ALL
                QUERY←PUNCH←ALL
                QUERY←HOLD
                CHANGE←SYSTEM←CLASS←ALL

transform OP2(Command: CommandName)

refcond Command<:Cat2

effect  N"PendingRequests = PendingRequests ||
              S"((NewMsgId,
                 OpRequest,
                 Command,
                   S"R:ResponseSlot(E"N:ProcessName
                        (N<:CurrentNkcps &
                         R = (N,nil,NoResponse)))))
        &
        A"N:ProcessName (N<:CurrentNkcps ->
            KernelCalled(SendMessage(N)))
```

```
OP3:            Single message sent
                Mapping: User id -> Nkcp id
                Single response expected
                No device state information modifications

        commands:
                QUERY←FILES←USERID
                QUERY←READER←USERID
                QUERY←PRINTER←USERID
                QUERY←PUNCH←USERID
                CHANGE←USERID
                FREE
                HOLD
                ORDER←USERID
                PURGE←USERID

transform OP3(Command: CommandName)

refcond Command<:Cat3

effect  N"PendingRequests = PendingRequests ||
              S"((NewMsgId,
                  MapUserId,
                  Command,
                  S"((AuthProcess,
                      nil,
                      NoResponse))))
        &
        KernelCalled(SendMessage(AuthProcess))
```

```
OP4a:    commands:
                 BACKSPAC
                 FLUSH
                 REPEAT

transform OP4a(Command: CommandName,
                 Raddr: DeviceAddress)

refcond Command<:Cat4
                 &
                 E"P:PrinterEntry (P<:Printers &
                     P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                 (~(P.State ~= NotSpooling
                     &
                     P.CyclePosition = AttachedToSpoolingProcess)
                 => Error <>
              N"PendingRequests = PendingRequests ||
                S"((NewMsgId,
                    OpRequest,
                    Command,
                      S"((P.AttachedProcess,
                        nil,
                        NoResponse))))
              &
                 KernelCalled(SendMessage(P.AttachedProcess))))
```

```
OP4b:    commands:
                 BACKSPAC
                 FLUSH
                 REPEAT

transform OP4b(Command: CommandName,
                 Raddr: DeviceAddress)

refcond Command<:Cat4
                 &
                 E"P:PunchEntry (P<:Punches &
                     P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                 (~(P.State ~= NotSpooling
                     &
                     P.CyclePosition = AttachedToSpoolingProcess)
                 => Error <>
             N"PendingRequests = PendingRequests ||
               S"((NewMsgId,
                  OpRequest,
                  Command,
                     S"((P.AttachedProcess,
                       nil,
                       NoResponse))))
             &
                 KernelCalled(SendMessage(P.AttachedProcess))))
```

```
OPS:    command:
                SPACE

transform OPS(Command: CommandName,
                Raddr: DeviceAddress)

refcond Command = SPACE
                &
                E"P:PrinterEntry (P<:Printers &
                    P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                (~(P.State ~= NotSpooling
                    &
                    P.CyclePosition = AttachedToSpoolingProcess)
                => Error <>
            N"PendingRequests = PendingRequests ||
              S"((NewMsgId,
                  OpRequest,
                  Command,
                    S"((P.AttachedProcess,
                      nil,
                      NoResponse))))
            &
                KernelCalled(SendMessage(P.AttachedProcess))))
```

```
OP6:    commands:
                 ORDER←SYSTEM
                 PURGE←SYSTEM

transform OP6(Command: CommandName,
                 Nkcps: set of ProcessName)

refcond Command<:CommandName
                 &
                 A"N:ProcessName(N<:Nkcps ->
                     E"CN:NkcpEntry
                          (CN<:CurrentNkcps
                           &
                           CN.Process = N))

effect  N"PendingRequests = PendingRequests ||
               S"((NewMsgId,
                   OpRequest,
                   Command,
                   S"R:ResponseSlot(E"N:ProcessName
                      (N<:Nkcps
                       &
                       R = (N,nil,NoResponse)))))
        &
                 A"N:ProcessName (N<:Nkcps ->
                     KernelCalled(SendMessage(N)))
```

```
OP7a:   LOCATE of Reader

transform OP7a(Command: CommandName,
               Raddr: DeviceAddress)

refcond Command = LOCATE←RADDR
               &
               E"R:ReaderEntry (R<:Readers &
                   R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
               (R.CyclePosition ~<:S"(AttachedToSpoolingProcess,
                                      AttachedToUser)
               => Error <>
            N"PendingRequests = PendingRequests ||
              S"((NewMsgId,
                 OpRequest,
                 Command,
                   S"((R.AttachedProcess,
                     nil,
                     NoResponse))))
            &
               KernelCalled(SendMessage(D.AttachedProcess))))
```

OP7b: LOCATE of Printer

```
transform OP7b(Command: CommandName,
               Raddr: DeviceAddress)

refcond Command = LOCATE-RADDR
               &
               E"P:PrinterEntry (P<:Printers &
                   P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
               (P.CyclePosition ~<:S"(AttachedToSpoolingProcess,
                                      AttachedToUser)
               => Error <>
            N"PendingRequests = PendingRequests ||
              S"((NewMsgId,
                 OpRequest,
                 Command,
                   S"((P.AttachedProcess,
                     nil,
                     NoResponse))))
            &
               KernelCalled(SendMessage(P.AttachedProcess))))
```

```
OP7c:    LOCATE of Punch

transform OP7c(Command: CommandName,
               Raddr: DeviceAddress)

refcond Command = LOCATE←RADDR
                  &
                  E"P:PunchEntry (P<:Punches &
                      P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                  (P:CyclePosition ~<:S"(AttachedToSpoolingProcess,
                                          AttachedToUser)
                  => Error <>
              N"PendingRequests = PendingRequests ||
                S"((NewMsgId,
                    OpRequest,
                    Command,
                      S"((P.AttachedProcess,
                        nil,
                        NoResponse))))
              &
                  KernelCalled(SendMessage(P.AttachedProcess))))
```

OP7d: LOCATE of Tape Drive

```
transform OP7d(Command: CommandName,
                Raddr: DeviceAddress)

refcond Command = LOCATE←RADDR
                &
                E"T:TapeDriveEntry (T<:TapeDrives &
                    T.Raddr = Raddr)

effect  E"T:TapeDriveEntry(T<:TapeDrives & T.Raddr = Raddr &
                (T.State ~= AttachedToUser
                => Error <>
            N"PendingRequests = PendingRequests ||
              S"((NewMsgId,
                  OpRequest,
                  Command,
                    S"((T.AttachedProcess,
                      nil,
                      NoResponse))))
            &
                KernelCalled(SendMessage(T.AttachedProcess))))
```

```
OP8:    command:
                 SHUTDOWN

transform OP8(Command: CommandName)

refcond Command = SHUTDOWN

effect  N"ShuttingDown = true
        &
        N"PrinterSpoolRequests = PrinterSpoolRequests --
                                   S"DR:OutputDeviceEntry(
                                        DR.State = WaitingForDevice)
        &
        N"PunchSpoolRequests = PunchSpoolRequests --
                                 S"DR:OutputDeviceEntry(
                                      DR.State = WaitingForDevice)
```

OP9a: Vary offline (Reader)

```
transform OP9a(Command: CommandName,
               Raddr: DeviceAddress)

refcond Command = VARY←OFFLINE
               &
               E"R:ReaderEntry (R<:Readers &
                   R.Raddr = Raddr)

effect  E"P:ReaderEntry(R<:Readers & R.Raddr = Raddr &
               (R.CyclePosition ~<:S"(Availa[illegible]e,OffLine)
               => Error <>
          N"Readers = Readers ~~ S"(R) ||
            S"((R.Raddr,
       /*##*/    NotSpooling,
       /*##*/    OffLine,
                 R.AttachedProcess,
                 R.ClassesServedCurrently,
                 R.ClassesServedNextCycle,
                 R.ChannelStatusWord,
                 R.LineBuffer))
          &
       KernelCalled(SendMessage(OpProcess))))
```

OP9b: Vary offline (Printer)

```
transform OP9b(Command: CommandName,
                Raddr: DeviceAddress)

refcond Command = VARY-OFFLINE
                &
                E"P:PrinterEntry (P<:Printers &
                P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                (P.CyclePosition ~<:S"(Available,OffLine)
                => Error <>
           N"Printers = Printers ~~ S"(P) ||
             S"((P.Raddr,
        /*##*/    NotSpooling,
        /*##*/    OffLine,
                  P.AttachedProcess,
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
                  P.RelinquishDeviceRequestState,
                  P.ChannelStatusWord))
           &
        KernelCalled(SendMessage(OpProcess))))
```

OP9c: Vary offline (Punch)

```
transform OP9c(Command: CommandName,
                Raddr: DeviceAddress)

refcond Command = VARY-OFFLINE
                &
                E"P:PunchEntry (P<:Punches &
                    P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                (P.CyclePosition ~<:S"(Available,OffLine)
                => Error <>
            N"Punches = Punches ~~ S"(P) ||
              S"((P.Raddr,
        /*##*/    NotSpooling,
        /*##*/    OffLine,
                  P.AttachedProcess,
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
                  P.RelinquishDeviceRequestState,
                  P.ChannelStatusWord!)
            &
        KernelCalled(SendMessage(OpProcess))))
```

OP9d: Vary offline (Tape Drive)

```
transform OP9d(Command: CommandName,
               Raddr: DeviceAddress)

refcond Command = VARY←OFFLINE
               &
               E"T:TapeDriveEntry (T<:TapeDrives &
                   T.Raddr = Raddr)

effect  E"T:TapeDriveEntry(T<:TapeDrives & T.Raddr = Raddr &
               (T.State ~<:S"(Available,OffLine)
               => Error <>
            N"TapeDrives = TapeDrives ~~ S"(T) ||
              S"((T.Raddr,
        /*##*/     OffLine,
                   T.AttachedProcess))
            &
        KernelCalled(SendMessage(OpProcess))))
```

OP10a: Vary online (Reader)

```
transform OP10a(Command: CommandName,
                Raddr: DeviceAddress)

refcond Command = VARY←ONLINE
                &
                E"R:ReaderEntry (R<:Readers &
                    R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                (~(R.State = NotSpooling
                    &
                    R.CyclePosition = OffLine)
                => Error <>
            N"Readers = Readers ~~ S"(R) ||
              S"((R.Raddr,
        /*##*/    Drained,
        /*##*/    Available,
        /*##*/    URProcess,
        /*##*/    R.ClassesServedNextCycle,
                  R.ClassesServedNextCycle,
                  R.ChannelStatusWord,
                  R.LineBuffer))
            &
        KernelCalled(SendMessage(OpProcess))))
```

OP10b: Vary online (Printer)

```
transform OP10b(Command: CommandName,
                Raddr: DeviceAddress)

refcond Command = VARY-ONLINE
                &
                E"P:PrinterEntry (P<:Printers &
                    P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                (~(P.State = NotSpooling
                    &
                    P.CyclePosition = OffLine)
                => Error <>
            N"Printers = Printers ~~ S"(P) ||
              S"((P.Raddr,
        /*##*/    Drained,
        /*##*/    Available,
        /*##*/    URProcess,
        /*##*/    P.ClassesServedNextCycle,
                  P.ClassesServedNextCycle,
        /*##*/    NoNeed,
                  P.ChannelStatusWord))
            &
        KernelCalled(SendMessage(OpProcess))))
```

OP10c: Vary online (Punch)

```
transform OP10c(Command: CommandName,
                Raddr: DeviceAddress)

refcond Command = VARY←ONLINE
                &
                E"P:PunchEntry (P<:Punches &
                    P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                (~(P.State = NotSpooling
                    &
                    P.CyclePosition = OffLine)
                => Error <>
            N"Punches = Punches ~~ S"(P) ||
              S"((P.Raddr,
        /*##*/    Drained,
        /*##*/    Available,
        /*##*/    URProcess,
        /*##*/    P.ClassesServedNextCycle,
                  P.ClassesServedNextCycle,
        /*##*/    NoNeed,
                  P.ChannelStatusWord))
            &
        KernelCalled(SendMessage(OpProcess))))
```

OP10d: Vary online (Tape Drive)

```
transform OP10d(Command: CommandName,
                Raddr: DeviceAddress)

refcond Command = VARY←ONLINE
                &
                E"T:TapeDriveEntry(T<:TapeDrives &
                    T.Raddr = Raddr)

effect  E"T:TapeDriveEntry(T<:TapeDrives & T.Raddr = Raddr &
                (~(T.State = OffLine)
                => Error <>
            N"TapeDrives = TapeDrives ~~ S"(T) ||
              S"((T.Raddr,
        /*##*/    Available,
        /*##*/    URProcess))
            &
                   KernelCalled(SendMessage(OpProcess))))
```

AUTH3a: Attach device to process (request from AuthProcess) (Reader)

```
transform AUTH3a(Raddr: DeviceAddress,
                 Process: ProcessName)

refcond E"R:ReaderEntry (R<:Readers &
                         R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                  (A"N:NkcpEntry (N<:CurrentNkcps ->
                       N.Process ~= Process)
                  => Error <>
             N"Readers = Readers ~~ S"(R) ||
               S"((R.Raddr,
        /*##*/      (R.State=Drained
                      &
                      R.CyclePosition=Available
                      &
                      GrantedAccess=>
                           NotSpooling
                        <> R.State),
        /*##*/      (R.State=Drained
                      &
                      R.CyclePosition=Available
                      &
                      GrantedAccess=>
                           AttachedToUser
                        <> R.CyclePosition),
        /*##*/      (R.State=Drained
                      &
                      R.CyclePosition=Available
                      &
                      GrantedAccess=>
                           Process
                        <> R.AttachedProcess),
                    R.ClassesServedCurrently,
                    R.ClassesServedNextCycle,
                    R.ChannelStatusWord,
                    R.LineBuffer))
             &
                       (R.State=Drained
                      &
                      R.CyclePosition=Available->
                           KernelCalled(GrantAccess(R.Raddr)))
             &
                      KernelCalled(SendMessage(AuthProcess))))
```

AUTH3b: Attach device to process (request from AuthProcess) (Printer)

```
transform AUTH3b(Raddr: DeviceAddress,
                 Process: ProcessName)

refcond E"P:PrinterEntry (P<:Printers &
                    P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                 (A"N:NkcpEntry (N<:CurrentNkcps ->
                     N.Process ~= Process)
                 => Error <>
            N"Printers = Printers ~= S"(P) ||
              S"((P.Raddr,
        /*##*/     (P.State=Drained
                    &
                    P.CyclePosition=Available
                    &
                    GrantedAccess=>
                        NotSpooling
                     <> P.State),
        /*##*/     (P.State=Drained
                    &
                    P.CyclePosition=Available
                    &
                    GrantedAccess=>
                        AttachedToUser
                     <> P.CyclePosition),
        /*##*/     (P.State=Drained
                    &
                    P.CyclePosition=Available
                    &
                    GrantedAccess=>
                        Process
                     <> P.AttachedProcess),
                 P.ClassesServedCurrently,
                 P.ClassesServedNextCycle,
                 P.RelinquishDeviceRequestState,
                 P.ChannelStatusWord))
            &
                  (R.State=Drained
                  &
                  P.CyclePosition=Available->
                      KernelCalled(GrantAccess(P.Raddr)))
            &
                  KernelCalled(SendMessage(AuthProcess))))
```

AUTH3c: Attach device to process (request from AuthProcess) (Punch)

```
transform AUTH3c(Raddr: DeviceAddress,
                 Process: ProcessName)

refcond E"P:PunchEntry (P<:Punches &
                       P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                 (A"N:NkcpEntry (N<:CurrentNkcps ->
                      N.Process ~= Process)
                 => Error <>
            N"Punches = Punches ~~ S"(P) ||
              S"((P.Raddr,
      /*##*/      (P.State=Drained
                    &
                    P.CyclePosition=Available
                    &
                    GrantedAccess=>
                         NotSpooling
                     <> P.State),
      /*##*/      (P.State=Drained
                    &
                    P.CyclePosition=Available
                    &
                    GrantedAccess=>
                         AttachedToUser
                     <> P.CyclePosition),
      /*##*/      (P.State=Drained
                    &
                    P.CyclePosition=Available
                    &
                    GrantedAccess=>
                         Process
                     <> P.AttachedProcess),
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
                  P.RelinquishDeviceRequestState,
                  P.ChannelStatusWord))
            &
                    (R.State=Drained
                    &
                    P.CyclePosition=Available->
                         KernelCalled(GrantAccess(P.Raddr)))
            &
                    KernelCalled(SendMessage(AuthProcess))))
```

AUTH3d: Attach tape drive (request from AuthProcess)

```
transform AUTH3d(Raddr: DeviceAddress,
                 Process: ProcessName,
                 ReqAccess: AccessModes)

refcond E"T:TapeDriveEntry (T<:TapeDrives &
                 T.Raddr = Raddr)

effect  E"T:TapeDriveEntry(T<:TapeDrives & T.Raddr = Raddr &
                 (A"N:NkcpEntry (N<:CurrentNkcps ->
                      N.Process ~= Process)
                 => Error <>
             N"TapeDrives = TapeDrives ~ S"(T) ||
               S"((T.Raddr,
        /*##*/      (T.State=Available
                      &
                      GrantedAccess=>
                           AttachedToUser
                        <> T.State),
        /*##*/      (T.State=Available
                      &
                      GrantedAccess=>
                           Process
                        <> T.AttachedProcess)))
             &
                      KernelCalled(SendMessage(AuthProcess))))
```

OP11a: Attach device to process (request from operator) (Reader)

```
transform OP11a(Raddr: DeviceAddress,
                   Process: ProcessName)

refcond E"R:ReaderEntry (R<:Readers &
                        R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                   (A"N:NkcpEntry (N<:CurrentNkcps ->
                        N.Process ~= Process)
                   => Error <>
               N"Readers = Readers ~~ S"(R) ||
                 S"((R.Raddr,
         /*##*/     (R.State=Drained
                        &
                        R.CyclePosition=Available
                        &
                        GrantedAccess=>
                            NotSpooling
                         <> R.State),
         /*##*/     (R.State=Drained
                        &
                        R.CyclePosition=Available
                        &
                        GrantedAccess=>
                            AttachedToUser
                         <> R.CyclePosition),
         /*##*/     (R.State=Drained
                        &
                        R.CyclePosition=Available
                        &
                        GrantedAccess=>
                            Process
                         <> R.AttachedProcess),
                    R.ClassesServedCurrently,
                    R.ClassesServedNextCycle,
                    R.ChannelStatusWord,
                    R.LineBuffer))
               &
                   KernelCalled(SendMessage(OpProcess))
               &
                    (R.State=Drained
                     &
                     R.CyclePosition=Available->
                         (KernelCalled(GrantAccess(R.Raddr))
                          &
                          (GrantedAccess ->
                                KernelCalled(SendMessage(Process)))))))
```

OP11b: Attach device to process (request from operator) (Printer)

```
transform OP11b(Raddr: DeviceAddress,
                Process: ProcessName)

refcond E"P:PrinterEntry (P<:Printers &
                          P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                  (A"N:NkcpEntry (N<:CurrentNkcps ->
                       N.Process ~= Process)
                  => Error <>
             N"Printers = Printers ~~ S"(P) ||
               S"((P.Raddr,
        /*##*/     (P.State=Drained
                     &
                     P.CyclePosition=Available
                     &
                     GrantedAccess=>
                         NotSpooling
                      <> P.State),
        /*##*/     (P.State=Drained
                     &
                     P.CyclePosition=Available
                     &
                     GrantedAccess=>
                         AttachedToUser
                      <> P.CyclePosition),
        /*##*/     (P.State=Drained
                     &
                     P.CyclePosition=Available
                     &
                     GrantedAccess=>
                         Process
                      <> P.AttachedProcess),
                   P.ClassesServedCurrently,
                   P.ClassesServedNextCycle,
                   P.RelinquishDeviceRequestState,
                   P.ChannelStatusWord))
             &
                     KernelCalled(SendMessage(OpProcess))
             &
                      (P.State=Drained
                       &
                       P.CyclePosition=Available->
                           (KernelCalled(GrantAccess(P.Raddr))
                            &
                            (GrantedAccess ->
                                   KernelCalled(SendMessage(Process)))))))
```

OP11c: Attach device to process (request from operator) (Punch)

```
transform OP11c(Raddr: DeviceAddress,
                Process: ProcessName)

refcond E"P:PunchEntry (P<:Punches &
                        P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                (A"N:NkcpEntry (N<:CurrentNkcps ->
                    N.Process ~= Process)
                => Error <>
             N"Punches = Punches ~~ S"(P) ||
               S"((P.Raddr,
        /*##*/     (P.State=Drained
                    &
                    P.CyclePosition=Available
                    &
                    GrantedAccess=>
                        NotSpooling
                     <> P.State),
        /*##*/     (P.State=Drained
                    &
                    P.CyclePosition=Available
                    &
                    GrantedAccess=>
                        AttachedToUser
                     <> P.CyclePosition),
        /*##*/     (P.State=Drained
                    &
                    P.CyclePosition=Available
                    &
                    GrantedAccess=>
                        Process
                     <> P.AttachedProcess),
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
                  P.RelinquishDeviceRequestState,
                  P.ChannelStatusWord))
             &
                   KernelCalled(SendMessage(OpProcess))
             &
                   (P.State=Drained
                    &
                    P.CyclePosition=Available->
                        (KernelCalled(GrantAccess(P.Raddr))
                         &
                         (GrantedAccess ->
                                KernelCalled(SendMessage(Process))))))
```

OP12a: Detach dedicated device from user (request from operator) (Reader)

```
transform OP12a(Raddr: DeviceAddress,
                Process: ProcessName)

refcond E"R:ReaderEntry (R<:Readers &
                  R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                (~(R.State = NotSpooling
                   &
                   R.CyclePosition = AttachedToUser
                   &
                   R.AttachedProcess = Process)
                => Error <>
             N"Readers = Readers ~~ S"(R) ||
               S"((R.Raddr,
                   R.State,
        /*##*/     DetachPending,
                   R.AttachedProcess,
                   R.ClassesServedCurrently,
                   R.ClassesServedNextCycle,
                   R.ChannelStatusWord,
                   R.LineBuffer))
             &
             N"PendingRequests = PendingRequests ||
               S"((NewMsgId,
                     RelinquishDevice,
                     Undefined,
                     S"((R.AttachedProcess,
                        nil,
                        NoResponse))))))
```

OP12b: Detach dedicated device from user (request from operator) (Printer)

```
transform OP12b(Raddr: DeviceAddress,
                Process: ProcessName)

refcond E"P:PrinterEntry (P<:Printers &
                P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                (~(P.State = NotSpooling
                   &
                   P.CyclePosition = AttachedToUser
                   &
                   P.AttachedProcess = Process)
                => Error <>
            N"Printers = Printers ~~ S"(P) ||
              S"((P.Raddr,
                  P.State,
        /*##*/    DetachPending,
                  P.AttachedProcess,
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
                  P.RelinquishDeviceRequestState,
                  P.ChannelStatusWord))
            &
            N"PendingRequests = PendingRequests ||
              S"((NewMsgId,
                  RelinquishDevice,
                  Undefined,
                  S"((P.AttachedProcess,
                     nil,
                     NoResponse))))))
```

OP12c: Detach dedicated device from user (request from operator) (Punch)

```
transform OP12c(Raddr: DeviceAddress,
                Process: ProcessName)

refcond E"P:PunchEntry (P<:Punches &
                  P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                (~(P.State = NotSpooling
                   &
                   P.CyclePosition = AttachedToUser
                   &
                   P.AttachedProcess = Process)
                => Error <>
            N"Punches = Punches ~~ S"(P) ||
              S"((P.Raddr,
                  P.State,
        /*##*/    DetachPending,
                  P.AttachedProcess,
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
                  P.RelinquishDeviceRequestState,
                  P.ChannelStatusWord))
            &
            N"PendingRequests = PendingRequests ||
              S"((NewMsgId,
                   RelinquishDevice,
                   Undefined,
                   S"((P.AttachedProcess,
                     nil,
                     NoResponse)))))
```

NKCP2a: Detach dedicated device from user (request from attached process) (Reader)

```
transform NKCP2a(Raddr: DeviceAddress,
                 Process: ProcessName)

refcond E"R:ReaderEntry (R<:Readers &
                    R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                 (~(R.State = NotSpooling
                    &
                    R.CyclePosition = AttachedToUser
                    &
                    R.AttachedProcess = Process)
                 => Error <>
              N"Readers = Readers ~~ S"(R) ||
                S"((R.Raddr,
        /*##*/     (DeviceReleased=>
                         Drained
                      <> R.State),
        /*##*/     (DeviceReleased=>
                         Available
                      <> R.CyclePosition),
        /*##*/     (DeviceReleased=>
                         URProcess
                      <> R.AttachedProcess),
                   R.ClassesServedCurrently,
                   R.ClassesServedNextCycle,
                   R.ChannelStatusWord,
                   R.LineBuffer))
              &
                    KernelCalled(ReleaseDevice(R.Raddr))
              &
                    KernelCalled(SendMessage(R.AttachedProcess))
              &
                    KernelCalled(SendMessage(OpProcess))))
```

NKCP2b: Detach dedicated device from user (request from attached process) (Printer)

```
transform NKCP2b(Raddr: DeviceAddress,
                 Process: ProcessName)

refcond E"P:PrinterEntry (P<:Printers &
                 P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                 (~(P.State = NotSpooling
                     &
                     P.CyclePosition = AttachedToUser
                     &
                     P.AttachedProcess = Process)
                 => Error <>
              N"Printers = Printers ~~ S"(P) ||
                S"((P.Raddr,
        /*##*/      (DeviceReleased=>
                         Drained
                      <> P.State),
        /*##*/      (DeviceReleased=>
                         Available
                      <> P.CyclePosition),
        /*##*/      (DeviceReleased=>
                         URProcess
                      <> P.AttachedProcess),
                    P.ClassesServedCurrently,
                    P.ClassesServedNextCycle,
                    P.RelinquishDeviceRequestState,
                    P.ChannelStatusWord))
              &
                     KernelCalled(ReleaseDevice(P.Raddr))
              &
                     KernelCalled(SendMessage(P.AttachedProcess))
              &
                     KernelCalled(SendMessage(OpProcess))))
```

NKCP2c: Detach dedicated device from user (request from attached process) (Punch)

```
transform NKCP2c(Raddr: DeviceAddress,
                 Process: ProcessName)

refcond E"P:PunchEntry (P<:Punches &
                        P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                 (~(P.State = NotSpooling
                     &
                     P.CyclePosition = AttachedToUser
                     &
                     P.AttachedProcess = Process)
                 => Error <>
            N"Punches = Punches ~~ S"(P) ||
              S"((P.Raddr,
       /*##*/     (DeviceReleased=>
                        Drained
                     <> P.State),
       /*##*/     (DeviceReleased=>
                        Available
                     <> P.CyclePosition),
       /*##*/     (DeviceReleased=>
                        URProcess
                     <> P.AttachedProcess),
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
                  P.RelinquishDeviceRequestState,
                  P.ChannelStatusWord))
            &
                    KernelCalled(ReleaseDevice(P.Raddr))
            &
                    KernelCalled(SendMessage(P.AttachedProcess))
            &
                    KernelCalled(SendMessage(UpProcess))))
```

OP12d: Detach dedicated device from user (request from operator) (Tape Drive)

```
transform OP12d(Raddr: DeviceAddress,
                Process: ProcessName)

refcond E"T:TapeDriveEntry(T<:TapeDrives &
                T.Raddr=Raddr)

effect  E"T:TapeDriveEntry(T<:TapeDrives & T.Raddr = Raddr &
              (~(T.State=Attached
                  &
                  T.AttachedProcess=Process)
              => Error <>
          N"TapeDrives = TapeDrives ~~ S"(T) ||
            S"((T.Raddr,
        /*##*/    DetachPending,
                  T.AttachedProcess))
          &
          N"PendingRequests = PendingRequests ||
            S"((NewMagId,
                  RelinquishDevice,
                  Undefined,
                  S"((T.AttachedProcess,
                     nil,
                     NoResponse))))))
```

NKCP2d: Detach dedicated device from user (request from attached process) (Tape Drive)

```
transform NKCP2d(Raddr: DeviceAddress,
                 Process: ProcessName)

refcond E"T:TapeDriveEntry (T<:TapeDrives &
                     T.Raddr = Raddr)

effect  E"T:TapeDriveEntry(T<:TapeDrives & T.Raddr = Raddr &
                 (~(T.State = Attached
                     &
                     T.AttachedProcess = Process)
                  => Error <>
              N"TapeDrives = TapeDrives ~~ S"(T) ||
                S"((T.Raddr,
      /*##*/        (DeviceReleased=>
                           Available
                        <> T.State),
      /*##*/        (DeviceReleased=>
                           URProcess
                        <> T.AttachedProcess)))
              &
                 KernelCalled(ReleaseDevice(T.Raddr))
                 &
                 KernelCalled(SendMessage(T.AttachedProcess))
                 &
                 KernelCalled(SendMessage(OpProcess))))
```

NKCP1a: (Reader) Process message, relinquishing device as requested

```
transform NKCP1a(Raddr:DeviceAddress,
                 Process:ProcessName)

refcond E"R:ReaderEntry (R<:Readers &
                  R.Raddr = Raddr)

effect  E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                 (~(R.State = NotSpooling
                    &
                    R.CyclePosition = DetachPending
                    &
                    R.AttachedProcess = Process)
                 => Error <>
          N"Readers = Readers ~~ S"(R) ||
             S"((R.Raddr,
/*##*/         (DeviceReleased=>
                    Drained
                 <> R.State),
/*##*/         (DeviceReleased=>
                    Available
                 <> R.CyclePosition),
/*##*/         (DeviceReleased=>
                    URProcess
                 <> R.AttachedProcess),
               R.ClassesServedCurrently,
               R.ClassesServedNextCycle,
               R.ChannelStatusWord,
               R.LineBuffer))
        &
             KernelCalled(ReleaseDevice(R.Raddr))
             &
             KernelCalled(SendMessage(R.AttachedProcess))
             &
             KernelCalled(SendMessage(OpProcess))))
```

NKCP1b: Process message, relinquishing device as requested (Printer)

```
transform NKCP1b(Raddr:DeviceAddress,
                Process:ProcessName)

refcond E"P:PrinterEntry (P<:Printers &
                    P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                (~(P.State = NotSpooling
                    &
                    P.CyclePosition = DetachPending
                    &
                    P.AttachedProcess = Process)
                => Error <>
          N"Printers = Printers ~~ S"(P) ||
            S"((P.Raddr,
        /*##*/      (DeviceReleased=>
                        Drained
                    <> P.State),
        /*##*/      (DeviceReleased=>
                        Available
                    <> P.CyclePosition),
        /*##*/      (DeviceReleased=>
                        URProcess
                    <> P.AttachedProcess),
                P.ClassesServedCurrently,
                P.ClassesServedNextCycle,
                P.RelinquishDeviceRequestState,
                P.ChannelStatusWord))
          &
              KernelCalled(ReleaseDevice(P.Raddr))
              &
              KernelCalled(SendMessage(P.AttachedProcess))
              &
              KernelCalled(SendMessage(OpProcess))))
```

NKCP1c: Process message, relinquishing device as requested (Punch)

```
transform NKCP1c(Raddr:DeviceAddress,
               Process:ProcessName).

refcond E"P:PunchEntry (P<:Punches &
                  P.Raddr = Raddr)

effect  E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
               (~(P.State = NotSpooling
                  &
                  P.CyclePosition = DetachPending
                  &
                  P.AttachedProcess = Process)
               => Error <>
           N"Punches = Punches ~~ S"(P) ||
             S"((P.Raddr,
    /*##*/       (DeviceReleased=>
                      Drained
                   <> P.State),
    /*##*/       (DeviceReleased=>
                      Available
                   <> P.CyclePosition),
    /*##*/       (DeviceReleased=>
                      URProcess
                   <> P.AttachedProcess),
                 P.ClassesServedCurrently,
                 P.ClassesServedNextCycle,
                 P.RelinquishDeviceRequestState,
                 P.ChannelStatusWord))
           &
               KernelCalled(ReleaseDevice(P.Raddr))
               &
               KernelCalled(SendMessage(P.AttachedProcess))
               &
               KernelCalled(SendMessage(OpProcess))))
```

OP13: Loadbuf

```
transform OP13(Raddr: DeviceAddress)

refcond E"P:PrinterEntry (P<:Printers&
                  P.Raddr = Raddr)

effect  E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
              (~(P.State = Drained
                  &
                  P.CyclePosition = Available)
              => Error <>
                  true))
```

NKCP1d: Process message, relinquishing device as requested (Tape Drive)

```
transform NCKP1d(Raddr: DeviceAddress,
                 Process: ProcessName)

refcond E"T:TapeDriveEntry (T<:TapeDrives &
                     T.Raddr = Raddr)

effect  E"T:TapeDriveEntry(T<:TapeDrives & T.Raddr = Raddr &
                 (~(T.State = DetachPending
                     &
                     T.AttachedProcess = Process)
                 => Error <>
             N"TapeDrives = TapeDrives ~~ S"(T) ||
               S"((T.Raddr,
         /*##*/     (DeviceReleased=>
                         Available
                      <> T.State)
         /*##*/     (DeviceReleased=>
                         URProcess
                      <> T.AttachedProcess)))
             &
                 KernelCalled(ReleaseDevice(T.Raddr))
                 &
                 KernelCalled(SendMessage(T.AttachedProcess))
                 &
                 KernelCalled(SendMessage(OpProcess))))
```

OP11d: Attach tape drive (request from operator)

```
transform OP11d(Raddr: DeviceAddress,
                ReqAccess: AccessModes,
                TapeSecLevel: ProcessName,
                Process: ProcessName)

refcond E"T:TapeDriveEntry (T<:TapeDrives&
                    T.Raddr = Raddr)

effect  E"T:TapeDriveEntry(T<:TapeDrives & T.Raddr = Raddr &
                (A"N:NkcpEntry (N<:CurrentNkcps ->
                    N.Process ~= Process)
                => Error <>
                    E"N:NkcpEntry(N<:CurrentNkcps &
                        N.Process=Process
                    &
            N"TapeDrives = TapeDrives ~~ S"(T) ||
              S"((T.Raddr,
        /*##*/     (T.Raddr<:N.UsableTapeDrives
                    &
                    T.State=Available
                    &
                    CheckedSecLevel
                    &
                    GrantedAccess=>
                        Attached
                     <> T.State),
        /*##*/     (T.Raddr<:N.UsableTapeDrives
                    &
                    T.State=Available
                    &
                    CheckedSecLevel
                    &
                    GrantedAccess=>
                        N.Process
                     <> T.AttachedProcess)))
            &
```

```
                      KernelCalled(SendMessage(OpProcess))
           &
                      (T.Raddr<:N.UsableTapeDrives
                       &
                       T.State = Available)->
                          (KernelCalled(CheckSecLevel)
                           &
                           (CheckedSecLevel->
                               (KernelCalled(GrantAccess(T.Raddr))
                                &
                                (GrantedAccess->
                                   KernelCalled(SendMessage(N.Process))))))))
```

AUTH1: Add Nkcp

```
transform AUTH1(Process: ProcessName
                Readers: set of DeviceAddress,
                Printers: set of DeviceAddress,
                Punches: set of DeviceAddress,
                TapeDrives: set of DeviceAddress)

refcond true

effect  (E"N:NkcpEntry (N<:CurrentNkcps &
                  N.Process = Process)
                => Error <>
                    N"CurrentNkcps = CurrentNkcps ||
                      S"((Process,
                          Readers,
                          Printers,
                          Punches,
                          TapeDrives)))
```

AUTH2: Delete Nkcp

```
transform AUTH2(Process: ProcessName)

effect   (A"N:NkcpEntry (N<:CurrentNkcps ->
                    N.Process ~= Process)
         |
         E"DR:OutputDeviceRequest
                 ((DR<:PrinterSpoolRequests
                   |
                   DR<:PunchSpoolRequests)
                 &
                 DR.Process = Process
                 &
                 DR.State = Processing)
         |
         E"R:ReaderEntry(R<:Readers &
            R.AttachedProcess = Process
            &
            R.CyclePosition<:S"(AttachedToSpoolingProcess,AttachedToUser))
         |
         E"P:PrinterEntry(P<:Printers &
            P.AttachedProcess = Process
            &
            P.CyclePosition<:S"(AttachedToSpoolingProcess,AttachedToUser))
         |
         E"P:PunchEntry(P<:Punches &
            P.AttachedProcess = Process
            &
            P.CyclePosition<:S"(AttachedToSpoolingProcess,AttachedToUser))
         |
         E"T:TapeDriveEntry(T<:TapeDrives &
             T.AttachedProcess = Process
             &
             T.State = AttachedToUser)
                 => Error <>
                     E"N:NkcpEntry(N<:CurrentNkcps & N.Process = Process &
                     N"CurrentNkcps = CurrentNkcps ~~S"(N)
             &
                     N"PrinterSpoolRequests = PrinterSpoolRequests --
                            S"DR:OutputDeviceEntry
                          (DR<:PrinterSpoolRequests
                           &
                          DR.AttachedProcess = N.Process)
             &
                     N"PunchSpoolRequests = PunchSpoolRequests --
                           S"DR:OutputDeviceRequest
                          (DR<:PunchSpoolRequests
                           &
                           DR.AttachedProcess = N.Process)))
```

KERN1: message from Kernel, re device availability (during scan at system initialization)

```
transform KERN1(Raddr: DeviceAddress)

effect (E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr) =>
        E"R:ReaderEntry(R<:Readers & R.Raddr = Raddr &
                        (R.State = Drained =>
           N"Readers = Readers ~~ S"(R) ||
              S"((R.Raddr,
        /*##*/    NotAvailableForSpooling
        /*##*/    OffLine,
                  R.AttachedProcess,
                  R.ClassesServedCurrently,
                  R.ClassesServedNextCycle,
                  R.ChannelStatusWord,
                  R.LineBuffer))
                        <> Error))
                   <> E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr) =>
        E"P:PrinterEntry(P<:Printers & P.Raddr = Raddr &
                        (P.State = Drained =>
           N"Printers = Printers ~~ S"(P) ||
              S"((P.Raddr,
        /*##*/    NotAvailableForSpooling,
        /*##*/    OffLine,
                  P.AttachedProcess,
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
                  P.RelinquishDeviceRequestState,
                  P.ChannelStatusWord))
                        <> Error))
                    <> E"P:PunchEntry(P<:Punches & P.Raddr = Raddr) =>
        E"P:PunchEntry(P<:Punches & P.Raddr = Raddr &
                        (P.State = Drained =>
           N"Punches = Punches ~~ S"(P) ||
              S"((P.Raddr,
        /*##*/    NotAvailableForSpooling,
        /*##*/    OffLine,
                  P.AttachedProcess,
                  P.ClassesServedCurrently,
                  P.ClassesServedNextCycle,
                  P.RelinquishDeviceRequestState,
                  P.ChannelStatusWord))
                        <> Error))
                   <> E"T:TapeDriveEntry(T<:TapeDrives & T.Raddr = Raddr) =>
        E"T:TapeDriveEntry(T<:TapeDrives & T.Raddr = Raddr &
                        (T.State = Available =>
           N"TapeDrives = TapeDrives ~~ S"(T) ||
              S"((T.Raddr,
        /*##*/    OffLine,
                  T.AttachedProcess))
                        <> Error)))
end URProcess
```

3.3.1: Authorization Process Informal Description

This section contains the informal description of the Authorization Process of KVM/370.

Overview

The Authorization Process performs several functions of major importance to the system security of KVM/370. The Authorization Process acts as the security policeman of the system by controlling accessibility to the machine via the logon/off protocols, and by helping the Kernel keep the system in a set of security-preserving states by controlling non-unit record device attachment for all system processes. It also examines the security conditions surrounding all device attachments (for both unit record and non-unit record devices) to NKCPs.

The Authorization Process performs the following major functions:

- logon sequence;
- logoff sequence;
- links;
- non-unit record device attachment for all other system processes;
- security condition assurance for all device attachments; and
- NKCP creation and deletion.

The Authorization Process monitors the status of all non-unit record devices in its device tables. It uses a data structure called the 'directory' to control both user access to the system and the processing of links.

Figures four and five display the organization of the transforms involved in the logon protocol. Figure six displays the various states in which a particular line can exist, and the legal transitions between states.

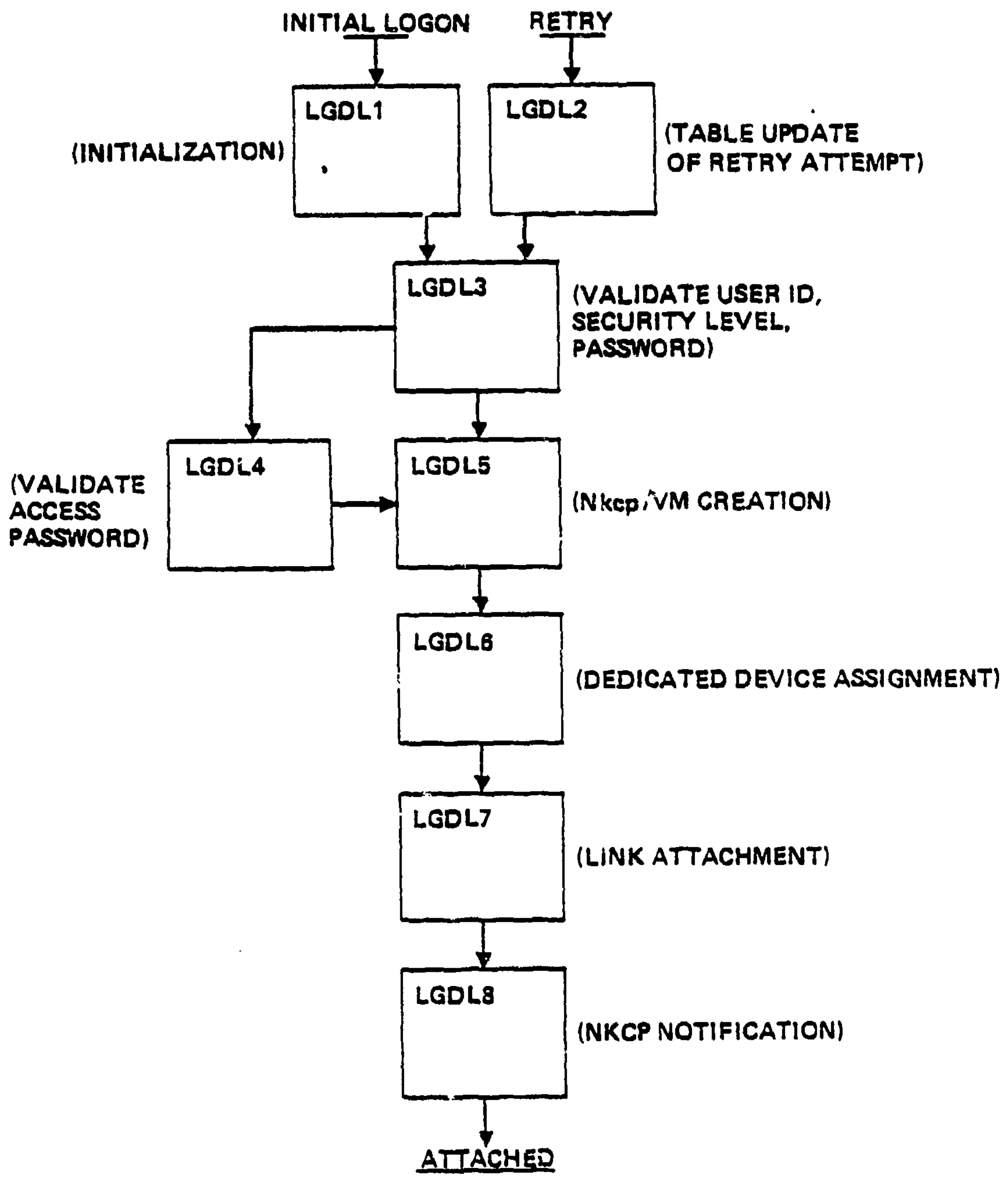

Figure 4

THE ATTACH VALIDATION SEQUENCE
(PROCESS STRUCTURE)

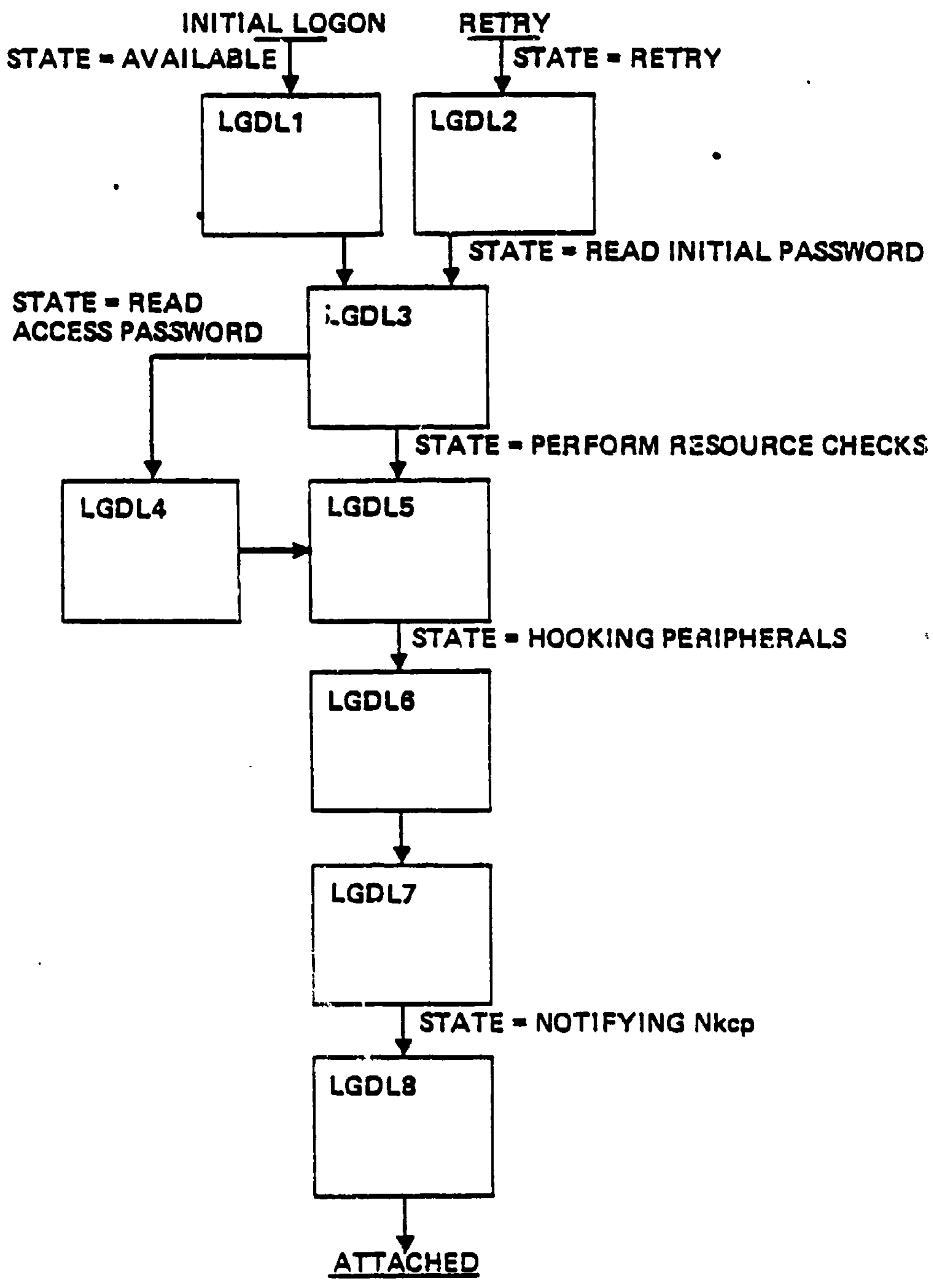

Figure 5
The Attach Validation Sequence
(State Transitions)

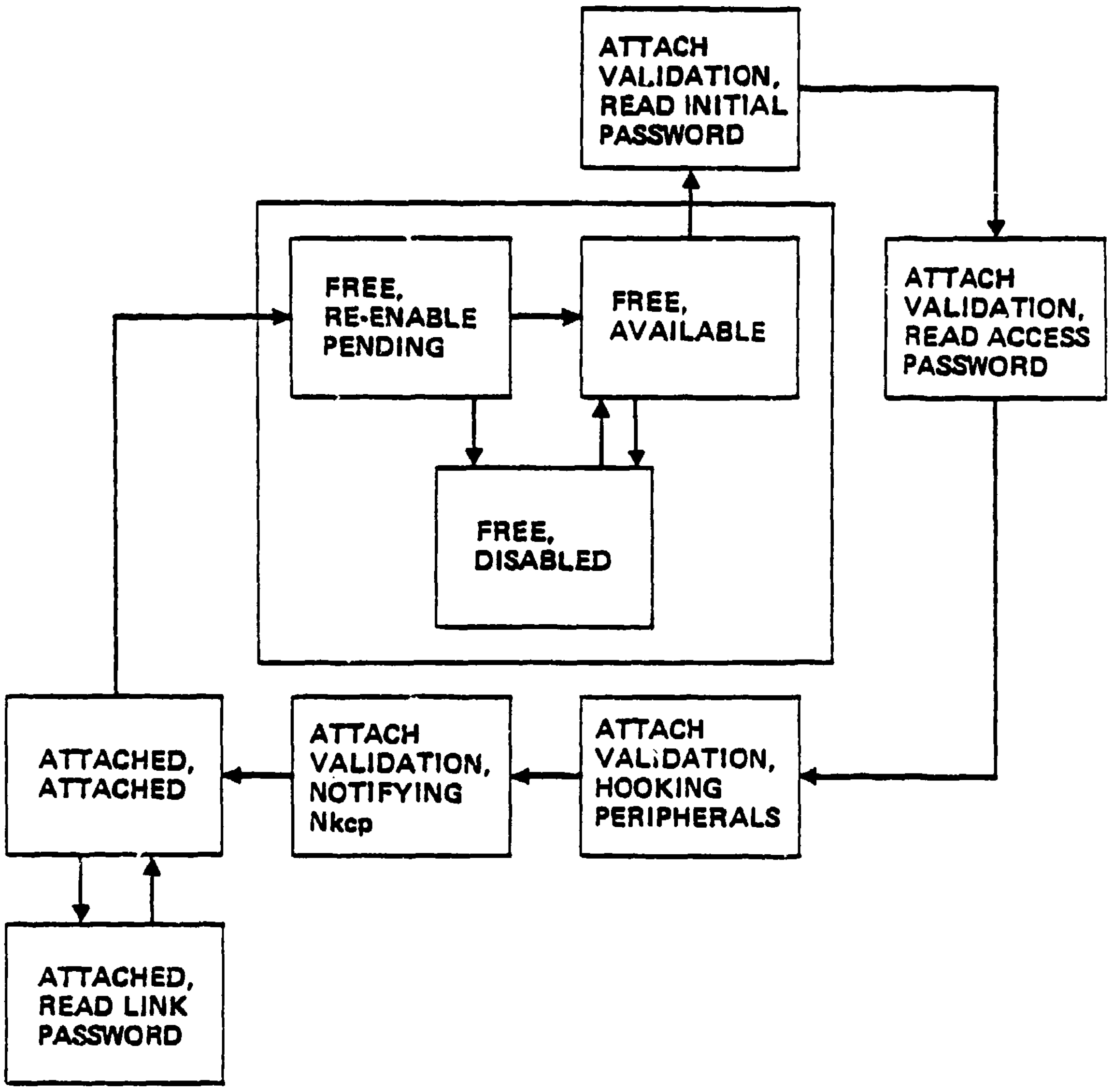
ATTACH VALIDATION, READ INITIAL PASSWORD
FREE, RE-ENABLE PENDING
FREE, AVAILABLE
ATTACH VALIDATION, READ ACCESS PASSWORD
FREE, DISABLED
ATTACHED, ATTACHED
ATTACH VALIDATION, NOTIFYING Nkcp
ATTACH VALIDATION, HOOKING PERIPHERALS
ATTACHED, READ LINK PASSWORD

Figure 6

The Line Activity Cycle

The Logon Sequence

When the user presses the 'break' key after dialing into the system, the resulting interrupt is reflected by the semi-trusted process monitoring all input lines to the Authorization Process, which then requests a 'read' on the line. The user then types the logon information, consisting of at least a header string, a user id, a password, and a security level. The Authorization Process uses this information and the directory information associated with this user to determine whether the attempted logon will be allowed. The factors involved in this decision include:

- the user's security level provided in the logon line;
- the communication line's security level;
- the security levels associated with the user in the user's directory entry; and
- whether an NKCP controlling the requested security level currently exists in the system, the number of NKCPs currently extant, and the maximum allowable number.

The Authorization Process must perform a fair amount of processing in order to determine whether the user is allowed to log on given the current state of the system and the requested security level.

Assuming the logon decision is affirmative, the Authorization Process uses the 'dedicated devices' section of the directory entry for this user to determine those devices that must be attached to (the controlling NKCP for) the new VM in order for the VM to perform. The Authorization Process attaches the permissible devices (pursuant to security conditions and the requested security level), the communication line and the new VM to the NKCP controlling this security level. These attachments all occur after first creating the NKCP if necessary and possible.

It then notifies the controlling NKCP of the existence of the new VM. Part of the message sent to the NKCP contains non-security-relevant directory information about the VM, in particular information about the VM's attached devices.

The Logoff Sequence

When a VM is to be purged from the system, the NKCP releases all the devices dedicated to the VM, performs accounting, and then informs the Authorization Process. It also provides the reason for the purge, which could be because the user logged off, or because the NKCP forced the user off (due to operator requests FORCE or SHUTDOWN), or because the user was disconnected and at least fifteen minutes have elapsed, etc.

The Authorization Process proceeds to destroy the VM and, if necessary, causes the communication line to be re-enabled for a new user connection.

If the Kernel does not allow the VM to be destroyed, the Authorization Process informs the NKCP of this fact.

NKCP Creation and Deletion

An NKCP is added to the system when an attempt is made to perform processing at a security level not currently being provided by the system. This request for support of a new security level may be caused by a user logging on or by an input spooling operation.

The Authorization Process attempts to create the NKCP to service the new security level. If it succeeds, the new user is attached in the normal manner or, in the case of an input spooling operation, the Unit Record Process is informed of the availability of the new NKCP so that it may attach the input device to the NKCP.

Due to limitations of Kernel table space or system performance requirements, creation of the new NKCP may not be possible. In this case the user's logon attempt is rejected or the input spooling operation is aborted.

It may also be the case that an NKCP has been created and the reason for its existence has disappeared. This will occur if the user does not complete the logon, the attempt to create a VM fails due to lack of table space, the input device becomes disabled or, in the case of an input spooling operation, the NKCP may not use the input device due to security conditions. The Authorization Process (or in the case of an input spooling operation, the Unit Record Process) will inform the NKCP that it is not needed. However, due to the inherent asynchronicity of the system, the NKCP may have become useful by one trusted process even as another was deciding it was not really needed. Thus the message to the NKCP should be interpreted as "Perhaps if you have no work to do, you may purge yourself," rather than an imperative command, "Purge yourself!" It is up to the NKCP to make the request of the Authorization Process which causes the actual purge of the NKCP.

Any time an NKCP discovers that it has no work to do (i.e., it has completed all of its tasks and nothing remains), it should request that it be purged from the system, by sending a message to the Authorization Process. The Authorization Process attempts to purge the NKCP. If the NKCP cannot be purged, the Authorization Process returns a message to the NKCP, which should then abort.

Operator Process
- Requests -

AUTOLOG

QUERY

    DASD -

    LINES -

    GRAF -

    SYSTEM -

    NAMES -

    USERS -

    user id -

    raddr - [raddr must be non-unit record device]

    ALL -

SHUTDOWN

ATTACH raddr - [raddr must be non-unit record device]

DETACH raddr - [raddr must be non-unit record device]

VARY raddr - [raddr must be non-unit record device]

LOCATE raddr - [raddr must be non-unit record device]

MapUserId

Unit Record Process
- Requests -

NeedNkcp

MapUserId

Unit Record Process
- Responses -

URDeviceAttached

URDeviceNotAvailable

NKCP Requests

Logoff

Disconnect

Detach raddr - [raddr must be non-unit record device]

DestroyMe

NKCP Responses

ResponseToOpRequest

3.3.2: Authorization Process Formal Specification

```
module AuthProcess
type
DeviceAddress,
LineAddress,
ProcessName,
VirtualMachineName,
VolumeId,

CommandName = (AUTOLOG, ATTACH←RADDR, DETACH←RADDR, VARY←ONLINE,
          VARY←OFFLINE, QUERY←DASD, QUERY←LINES, QUERY←GRAF,
          QUERY←NAMES, QUERY←USERS←X, QUERY←ALL, QUERY←SYSTEM←RADDR,
          QUERY←RADDR, QUERY←USERS←USERID, QUERY←USERID,
          LOCATE←RADDR, SHUTDOWN),

Cat6a = T"(QUERY←DASD, QUERY←LINES, QUERY←GRAF, QUERY←NAMES,
          QUERY←USERS←X, QUERY←ALL),

Cat6b = T"(QUERY←SYSTEM←RADDR, QUERY←RADDR),

Cat6c = T"(QUERY←USERS←USERID, QUERY←USERID),

MessageNames

constant
Dominates(ProcessName, ProcessName): boolean,
DeviceType(DeviceAddress): DeviceTypes,
#Cylinders(VolumeId): T"I:integer(0<I),
Raddr(String): DeviceAddress,
Laddr(String): DeviceAddress,
MsgName(String): MessageNames,
OpProcess, URProcess, NetworkProcess, AuthProcess: ProcessName,
TrustedProcesses: S"(OpProcess, URProcess, AuthProcess)

constant
AddressSpaceSize: T"I:integer(0<=I & I<=8191),
CodeSize: T"I:integer(0<=I & I<=8191),
Code: integer,
#MaxCylinders: T"I:integer(I>0),
#MaxRetries: T"I:integer(I>=0),
#MaxNkcps: T"I:integer(I>=0),
#MaxVMs: T"I:integer(I>=0)
```

```
type
RequestCategory = (Attach,ClearLine,ReDirectLine,WriteAndReadLine,
        OpRequest,NewVM,ConnectVM,NewUser,
        NewOrConnectedVM,RelinquishDevice),

ResponseStatus = (NoResponse,Responded),

AccessModes = (Read,Write),

LineStatus = (Retry,Disabled,Available,ReadInitialPassword,
        ReadAccessPassword,PerformResourceChecks,HookingPeripherals,
        NotifyingNkcp,Attached,ReadLinkPassword,ReEnablePending),

SharableDriveStatus = (OffLine,Available,AttachedToSystem),

DriveStatus = (OffLine,DetachPending,AttachedToUser,Available),

VolumeStatus = (Mounted,NotMounted),

LinkAccess = (R,RR,W,WR,M,MR,MW),

LineCondition = T"(Disabled,Available),

ActivityStatus = (Free,Attached,AttachValidation),

AccessCategory = (Logon,Dial),

ReasonTypes = (IncorrectLogon,ResourceFailure,SecurityViolation,
        MaxThresholdExceeded,NoNkcp,NoVM,TerminalClearanceMismatch),

LogoffReasons = (UserChoice,Forced,Disconnected),

DirectoryEntry = structure of (
        UserId = VirtualMachineName,
        LogonPassword = String,
        DialPassword = String,
        LinkPassword = String,
        MaxSecLevel = ProcessName,
        MinSecLevel = ProcessName,
        DedicatedDevices = set of DedicatedDeviceEntry,
        Links = set of MDLinkEntry,
        IplDefined = boolean,
        AccessPasswords = set of AccessPasswordEntry),
LineEntry = structure of (
          Laddr = LineAddress,
          MaxSecLevel = ProcessName,
          MinSecLevel = ProcessName,
          State = ActivityStatus,
          CyclePosition = LineStatus,
          RequestedSecLevel = ProcessName,
          AttachedVM = VirtualMachineName,
          Connection = AccessCategory,
          LineDropped = boolean,
          #Retries = 0..#MaxRetries,
          #AwaitingHooks = nonnegative integer,
          Msg = String),

NkcpEntry = structure of (
          Process = ProcessName,
          VMs = set of VMEntry,
          AttachedDevices = set of AttachedDeviceEntry,
          Links = set of MDLinkEntry),

AccessPasswordEntry = structure of (
          SecLevel = ProcessName,
          Password = String),

VMEntry =  structure of (
          VMName = VirtualMachineName,
          Laddr = LineAddress,
          Disconnected = boolean,
          Users = set of LineAddress),

DedicatedDeviceEntry = structure of (
          Raddr = DeviceAddress,
          VolSecLevel = ProcessName,
          Access = set of AccessModes),

AttachedDeviceEntry = structure of (
          Raddr = DeviceAddress,
          Access = set of AccessModes),

MDLinkEntry = structure of (
          MDName = MiniDiskName,
          Access = set of AccessModes),

ProcessLinkEntry = structure of (
          Process = ProcessName,
          Access = set of AccessModes),

URPOwnedDeviceEntry = structure of (
          Raddr = DeviceAddress,
          MaxSecLevel = ProcessName,
          MinSecLevel = ProcessName),
NonsharableDriveEntry = structure of (
        Raddr = DeviceAddress,
        MaxSecLevel = ProcessName,
        MinSecLevel = ProcessName,
        State = DriveStatus,
        AttachedProcess = ProcessName,
        Access = set of AccessModes),

SharableDriveEntry = structure of (
        Raddr = DeviceAddress,
        State = SharableDriveStatus,
        SecLevel = ProcessName,
        MountedVolume = VolumeId),

SharedVolumeEntry = structure of (
        Volume = VolumeId,
        SecLevel = ProcessName,
        MountedDevice = DeviceAddress,
        State = VolumeStatus),

MiniDiskEntry = structure of (
        MDName = MiniDiskName,
        ContainingVolume = VolumeId,
        Cylinders = T"I:integer(1<=I & I<=#MaxCylinders) ><
                        T"I:integer(1<=I & I<=#MaxCylinders),
        SecLevel = ProcessName,
        CurrentLinks = set of ProcessLinkEntry,
        AccessControlList = set of ACLEntry),

ACLEntry = structure of (
        User = VirtualMachineName,
        Access = set of AccessModes),

ResponseSlot = structure of (
        Respondent = ProcessName,
        Text = String,
        State = ResponseStatus),

PendingRequest = structure of (
        MsgId = MessageId,
        Kind = RequestCategory,
        Command = CommandName,
        Responses = set of ResponseSlot)
```

```
variable
#Nkcps: NkcpRange,
#VMs: VMRange,
#Users: UserRange,
ShuttingDown: boolean,
URPOwnedDevices: set of URPOwnedDeviceEntry,
NonsharableDrives: set of NonsharableDriveEntry,
SharableDrives: set of SharableDriveEntry,
SharedVolumes: set of SharedVolumeEntry,
MiniDisks: set of MiniDiskEntry,
CurrentNkcps: set of NkcpEntry,
Lines: set of LineEntry,
UserDirectory: set of DirectoryEntry,
PendingRequests: set of PendingRequest
```

```
initial
#Nkcps = 0
&
#VMs = 0
&
#Users = 0
&
(~ShuttingDown)
&
A"NS:NonsharableDriveEntry(NS<:NonsharableDrives ->
    NS.State = Available
    &
    NS.AttachedProcess = AuthProcess)
&
A"S:SharableDriveEntry(S<:SharableDrives ->
    S.State = Available)
&
A"V:SharedVolumeEntry(V<:SharedVolumes ->
    V.State = NotMounted)
&
A"M:MiniDiskEntry(M<:MiniDisks ->
   M.CurrentLinks = Empty)
&
CurrentNkcps = Empty
&
A"L:LineEntry(L<:Lines ->
    L.State = Free
    &
    L.CyclePosition = Available
    &
    L.AttachedVM = AuthProcess)
&
PendingRequests = Empty
```

```
invariant
#Nkcps <= #MaxNkcps
&
#VMs <= #MaxVMs
&
#Users <= #MaxUsers
&
InvariantsOfURPOwnedDevices
&
InvariantsOfNonsharableDrives
&
InvariantsOfSharableDrives
&
InvariantsOfSharedVolumes
&
InvariantsOfMiniDisks
&
InvariantsOfCurrentNkcps
&
InvariantsOfLines
&
InvariantsOfPendingRequests
&
InvariantsOfUserDirectory
```

```
InvariantsOfURPOwnedDevices =

A"U1,U2:URPOwnedDeviceEntry(
        U1<:URPOwnedDevices & U2<:URPOwnedDevices ->
    (U1.Raddr = U2.Raddr -> U1 = U2))
&
A"U:URPOwnedDeviceEntry(U<:URPOwnedDevices ->
     Dominates(U.MaxSecLevel,U.MinSecLevel)
     &
     DeviceType(Raddr) <: S"(Reader,Printer,Punch,TapeDrive))
```

```
InvariantsOfNonsharableDrives =

A"NS1,NS2:NonsharableDriveEntry(
          NS1<:NonsharableDrives & NS2<:NonsharableDrives ->
    (NS1.Raddr = NS2.Raddr -> NS1 = NS2))
&
A"NS:NonsharableDriveEntry(NS<:NonsharableDrives ->
     Dominates(NS.MaxSecLevel,NS.MinSecLevel)
     &
     (NS.State = Attached ->
         E"N:NkcpEntry(N<:CurrentNkcps &
             N.Process = NS.AttachedProcess
             &
             Dominates(NS.MaxSecLevel,N.Process)
             &
             Dominates(N.Process,NS.MinSecLevel)
             &
             E"A:AttachedDeviceEntry(A<:N.AttachedDevices &
                 A.Raddr = NS.Raddr))))
```

```
InvariantsOfSharableDrives =

A"S1,S2:SharableDriveEntry(S1<:SharableDrives & S2<:SharableDrives ->
    (S1.Raddr = S2.Raddr -> S1 = S2))
&
A"S:SharableDriveEntry(S<:SharableDrives ->
    (S.State = AttachedToSystem ->
         E"V:SharedVolumeEntry(V<:SharedVolumes &
              V.Volume = S.MountedVolume
              &
              V.State = Mounted
              &
              V.MountedDevice = S.Raddr
              &
              Dominates(S.SecLevel,V.SecLevel))))
```

```
InvariantsOfSharedVolumes =

A"V1,V2:SharedVolumeEntry(V1<:SharedVolumes & V2<:SharedVolumes ->
   (V1.Volume = V2.Volume -> V1 = V2))
&
A"V:SharedVolumeEntry(V<:SharedVolumes ->
   (V.State = Mounted ->
      E"S:SharableDriveEntry(S<:SharableDrives &
          S.Raddr = V.MountedDevice
          &
          S.MountedVolume = V.Volume
          &
          S.State = AttachedToSystem
          &
          Dominates(S.SecLevel,V.SecLevel))))
```

```
InvariantsOfMiniDisks =

A"M1,M2:MiniDiskEntry(M1<:MiniDisks & M2<:MiniDisks ->
    (M1.MDName = M2.MDName -> M1 = M2))
&
A"M:MiniDiskEntry(M<:MiniDisks ->
     E"V:SharedVolumeEntry(V<:SharedVolumes &
          V.Volume = M.ContainingVolume
          &
          Dominates(V.SecLevel,M.SecLevel))
     &
     M.Cylinders.2 > M.Cylinders.1
     &
     M.Cylinders.1 < #Cylinders(M.ContainingVolume)
     &
     M.Cylinders.2 <= #Cylinders(M.ContainingVolume)
     &
     A"C:ProcessLinkEntry(C<:M.CurrentLinks ->
          C.Access ~= Empty
          &
          E"N:NkcpEntry(N<:CurrentNkcps &
               N.Process = C.Process
               &
               E"L:MDLinkEntry(L<:N.Links &
                    L.MDName = M.MDName
                    &
                    L.Access = C.Access)
               &
               E"A:ACLEntry(A<:M.AccessControlList &
                    E"V:VMEntry(V<:N.VMs &
                         V.VMName = A.User))
               &
               (Write <: C.Access ->
                    N.Process = M.SecLevel)))
     &
     (M.CurrentLinks ~= Empty ->
          E"V:SharedVolumeEntry(V<:SharedVolumes &
               V.Volume = M.ContainingVolume
               &
               V.State = Mounted
               &
               E"S:SharableDriveEntry(S<:SharableDrives &
                    S.Raddr = V.MountedDevice
                    &
                    S.MountedVolume = V.Volume
                    &
                    S.State = AttachedToSystem)))
     &
A"A1,A2:ACLEntry(
     A1<:M.AccessControlList & A2<:M.AccessControlList ->
    (A1.User = A2.User -> A1 = A2))
&
A"A:ACLEntry(A<:M.AccessControlList ->
     E"D:DirectoryEntry(D<:UserDirectory &
          D.UserId = A.User)
     &
     A.Access ~= Empty))
```

```
InvariantsOfCurrentNkcps =

A"N1,N2:NkcpEntry(N1<:CurrentNkcps & N2<:CurrentNkcps ->
    (N1.Process = N2.Process -> N1 = N2))
&
A"N:NkcpEntry(N<:CurrentNkcps ->
    A"VM1,VM2:VMEntry(VM1<:N.VMs & VM2<:N.VMs ->
       (VM1.VMName = VM2.VMName -> VM1 = VM2))
    &
    A"AD1,AD2:AttachedDeviceEntry(
        AD1<:N.AttachedDevices & AD2<:N.AttachedDevices ->
       (AD1.Raddr = AD2.Raddr -> AD1 = AD2))
    &
    A"L1,L2:MDLinkEntry(L1<:N.Links & L2<:N.Links ->
       (L1.MDName = L2.MDName -> L1 = L2))
    &
    A"VM:VMEntry(VM<:N.VMs ->
        E"D:DirectoryEntry(D<:UserDirectory &
            D.UserId = V.VMName
            &
            Dominates(D.MaxSecLevel,N.Process)
            &
            Dominates(N.Process,D.MinSecLevel))
        &
        (VM.Disconnected ->
                A"L:LineEntry(L<:Lines ->
                    (L.AttachedVM = VM.VMName
                     &
                     L.RequestedSecLevel = N.Process
                     &
                     L.State = Attached) ->
                        L.Connection ~= Logon))
        &
        ((~VM.Disconnected) ->
                E"L:LineEntry(L<:Lines &
                    L.Laddr = VM.Laddr
                    &
                    L.AttachedVM = VM.VMName
                    &
                    L.RequestedSecLevel = N.Process
                    &
                    L.Connection = Logon
                    &
                    L.State <: S"(AttachValidation,Attached)))
        &
```

```
          A"U:LineAddress(U<:VM.Users ->
              E"L:LineEntry(L<:Lines &
                  L.Laddr = U
                  &
                  L.AttachedVM = VM.VMName
                  &
                  L.RequestedSecLevel = N.Process
                  &
                  L.Connection = Dial
                  &
                  L.State <: S"(AttachValidation,Attached))
              &
              U ~= VM.Laddr))
      &
      A"AD:AttachedDeviceEntry(AD<:N.AttachedDevices ->
          E"NS:NonsharableDriveEntry(NS<:NonsharableDrives &
              NS.Raddr = AD.Raddr
              &
              Dominates(NS.MaxSecLevel,N.Process)
              &
              Dominates(N.Process,NS.MinSecLevel))
          &
          AD.Access ~= Empty)
      &
      A"L:MDLinkEntry(L<:N.Links ->
          E"M:MiniDiskEntry(M<:MiniDisks &
              M.MDName = L.MDName
              &
              Dominates(N.Process,M.SecLevel)
              &
              E"C:ProcessListEntry(C<:M.CurrentLinks &
                  C.Process = N.Process
                  &
                  C.Access = L.Access)
              &
              E"A:ACLEntry(A<:M.AccessControlList &
                  E"VM:VMEntry(VM<:N.VMs &
                      VM.VMName = A.User))
              &
              (Write <: L.Access ->
                  M.SecLevel = N.Process))
          &
          L.Access ~= Empty))
  &
  A"N1,N2:NkcpEntry(N1<:CurrentNkcps & N2<:CurrentNkcps ->
          A"AD1,AD2:AttachedDeviceEntry(
            AD1<:N1.AttachedDevices & AD2<:N2.AttachedDevices ->
                  AD1.Raddr ~= Ad2.Raddr))
```

```
InvariantsOfLines =

A"L1,L2:LineEntry(L1<:Lines & L2<:Lines ->
   (L1.Laddr = L2.Laddr -> L1 = L2))
&
A"L:LineEntry(L<:Lines ->
    Dominates(L.MaxSecLevel,L.MinSecLevel)
    &
    (L.State = AttachValidation ->
        L.CyclePosition <: S"(Retry,ReadInitialPassword,
                              ReadAccessPassword,HookingPeripherals,
                              NotifyingNkcp))
    &
    (L.State = Attached ->
         L.CyclePosition <: S"(Attached,ReadLinkPassword))
    &
    (L.State = Free ->
        L.CyclePosition <: S"(Disabled,Available,ReEnablePending))
    &
    (L.State <: S"(AttachValidation,Attached) ->
         Dominates(L.MaxSecLevel,L.RequestedSecLevel)
         &
         Dominates(L.RequestedSecLevel,L.MinSecLevel))
    &
    (L.State = Attached ->
         E"N:NkcpEntry(N<:CurrentNkcps &
              N.Process = L.RequestedSecLevel
              &
              E"VM:VMEntry(VM<:N.VMs &
                   VM.VMName = L.AttachedVM
                   &
                   (L.Connection = Logon ->
                          VM.Laddr = L.Laddr
                          &
                          ~VM.Disconnected)
                   &
                   (L.Connection = Dial ->
                        E"U:LineAddress(U<:VM.Users &
                            U = L.Laddr))))))
&
```

```
A"L1,L2:LineEntry(L1<:Lines & L2<:Lines ->
     (L1.Raddr ~= L2.Raddr
      &
      L1.State = Attached
      &
      L2.State = Attached
      &
      L1.AttachedVM = L2.AttachedVM
      &
      L1.RequestedSecLevel = L2.RequestedSecLevel
      &
      L1.Connection = Logon) ->
         L2.Connection = Dial)
```

```
InvariantsOfPendingRequests =

A"P1,P2:PendingRequest(P1<:PendingRequests & P2<:PendingRequests ->
   (P1.MsgId = P2.MsgId -> P1 = P2))
&
A"P:PendingRequest(P<:PendingRequests ->
     A"R1,R2:ResponseSlot(R1<:P.Responses & R2<:P.Responses ->
         (R1.Respondent = R2.Respondent -> R1 = R2))
     &
     E"R:ResponseSlot(R<:P.Resnonses &
          R.State = NoResponse)
     &
     P.Responses ~= Empty)
```

```
InvariantsOfUserDirectory =

A"D1,D2:DirectoryEntry(D1<:UserDirectory & D2<:UserDirectory ->
   (D1.UserId = D2.UserId -> D1 = D2))
&
A"D:DirectoryEntry(D<:UserDirectory ->
    Dominates(D.MaxSecLevel,D.MinSecLevel)
    &
    A"DD1,DD2:DedicatedDeviceEntry(
        DD1<:D.DedicatedDevices & DD2<:D.DedicatedDevices ->
       (DD1.Raddr = DD2.Raddr -> DD1 = DD2))
    &
    A"DD:DedicatedDeviceEntry(DD<:D.DedicatedDevices ->
       (DeviceType(DD.Raddr) <: S"(Reader,Printer,Punch,TapeDrive) ->
                 E"U:URPOwnedDeviceEntry(U<:URPOwnedDevices &
                     DD.Raddr = U.Raddr
                     &
                     (DeviceType(DD.Raddr) = Reader ->
                         DD.VolSecLevel = nil
                         &
                         DD.Access = S"(Read))
                     &
                     (DeviceType(DD.Raddr) <: S"(Printer,Punch)->
                         DD.VolSecLevel = nil
                         &
                         DD.Access = S"(Write))
                     &
                     (DeviceType(DD.Raddr) = TapeDrive ->
                         Dominates(U.MaxSecLevel,DD.VolSecLevel)
                         &
                         Dominates(DD.VolSecLevel,U.MinSecLevel)
                         &
                         Dominates(D.MaxSecLevel,DD.VolSecLevel)
                         &
                         DD.Access ~= Empty)))
        &
        (DeviceType(DD.Raddr) ~<: S"(Reader,Printer,Punch,TapeDrive) ->
                 E"NS:NonsharableDriveEntry(NS<:NonsharableDrives &
                     NS.Raddr = DD.Raddr
                     &
                     Dominates(NS.MaxSecLevel,DD.VolSecLevel)
                     &
                     Dominates(DD.VolSecLevel,NS.MinSecLevel)
                     &
                     Dominates(D.MaxSecLevel,DD.VolSecLevel)
                     &
                     DD.Access ~= Empty)))
    &
A"L1,L2:MDLinkEntry(L1<:D.Links & L2<:D.Links ->
    (L1.MDName = L2.MDName -> L1 = L2))
&
A"L:MDLinkEntry(L<:D.Links ->
    E"M:MiniDiskEntry(M<:MiniDisks &
        M.MDName = L.MDName
        &
        E"A:ACLEntry(A<:M.AccessControlList &
            A.User = D.UserId
            &
            L.Access<<=A.Access)
        &
        Dominates(D.MaxSecLevel,M.SecLevel))
    &
    L.Access ~= Empty)
&
A"AP1,AP2:AccessPasswordEntry(
    AP1<:D.AccessPasswords & AP2<:D.AccessPasswords ->
    (AP1.SecLevel = AP2.SecLevel -> AP1 = AP2))
&
A"AP:AccessPasswordEntry(AP<:D.AccessPasswords ->
    Dominates(D.MaxSecLevel,AP.SecLevel)
    &
    Dominates(AP.SecLevel,D.MinSecLevel)))
```

NTWK1:   Network process message re line status (both request and response)

```
transform NTWK1(Laddr: LineAddress,
                CurrentLineStatus: LineCondition)

effect  (~(E"L:LineEntry(L<:Lines &
                    L.Laddr = Laddr))
        => Error  <>  NoError &
        E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
            N"Lines = Lines ~~ S"(L) ||
              S"((L.Laddr,
                  L.MaxSecLevel,
                  L.MinSecLevel,
        /*##*/     (L.State = Attached
                     |
                    (L.State = AttachValidation
                      &
                     L.CyclePosition~<:S"(HookingPeripherals,
                                           NotifyingNkcp)) =>
                       Free
                   <>  L.State),
        /*##*/     (L.State = Attached
                     |
                    (L.State = AttachValidation
                      &
                     L.CyclePosition~<:S"(HookingPeripherals,
                                           NotifyingNkcp)) =>
                       CurrentLineStatus
                   <>  L.CyclePosition),
                  L.RequestedSecLevel,
        /*##*/     (L.State = Attached
                     |
                    (L.State = AttachValidation
                      &
                     L.CyclePosition~<:S"(HookingPeripherals,
                                           NotifyingNkcp)) =>
                       AuthProcess
                   <>  L.AttachedVM),
                  L.Connection,
        /*##*/     (L.State = AttachValidation
                    &
                   L.CyclePosition<:S"(HookingPeripherals,NotifyingNkcp) =>
                       true
                   <>  L.LineDropped),
                  L.#Retries,
                  L.#AwaitingHooks,
                  L.Msg))
            &
```

```
              (L.State = Attached =>
                    E"N:NkcpEntry(
                      N<:CurrentNkcps & N.Process = L.AttachedProcess &
                    E"V:VMEntry(V<:N.VMs & V.VMName = L.AttachedVM &
                    N"CurrentNkcps = CurrentNkcps ~~ S"(N) ||
                      S"((N.Process,
                          N.VMs ~~ S"(V) ||
                            S"((V.VMName,
                                V.Laddr,
        /*##*/                  (L.Connection = Logon
                                 &
                                 L.Laddr = V.Laddr =>
                           true
                      <>  V.Disconnected),
        /*##*/                  (L.Connection = Dial
                                 &
                                 E"U:LineAddress(U<:V.Users &
                                   U = L.Laddr) =>
                          V.Users ~~ S"(L.Laddr)
                      <>  V.Users))),
                          N.AttachedDevices,
                          N.Links))))
                    &
                    KernelCalled(SendMessage(OpProcess))
              <> KernelCalled(SendMessage(Process)))))
```

LGDL1: Network Process message, LOGON or DIAL request received

```
transform LGDL1(Laddr: LineAddress,
                AttemptedCommand: AccessCategory,
                UserId: VirtualMachineName,
                RequestedSecLevel: ProcessName)

effect  (~(E"L:LineEntry(L<:Lines &
                   L.Laddr = Laddr
                   &
                   L.State = Free
                   &
                   L.CyclePosition = Available))
        => Error  <>  NoError &
        E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
            N"Lines = Lines ~~ S"(L) ||
            (Dominates(L.MaxSecLevel,RequestedSecLevel)
             &
             Dominates(RequestedSecLevel,L.MinSecLevel) =>
              S"((L.Laddr,
                  L.MaxSecLevel,
                  L.MinSecLevel,
        /*##*/    AttachValidation,
        /*##*/    ReadInitialPassword,
        /*##*/    RequestedSecLevel,
        /*##*/    UserId,
        /*##*/    AttemptedCommand,
        /*##*/    false,
        /*##*/    0,
        /*##*/    0,
        /*##*/    nil))
             <> S"((L.Laddr,
                  L.MaxSecLevel,
                  L.MinSecLevel,
        /*##*/    Free,
        /*##*/    ReEnablePending,
                  L.RequestedSecLevel,
        /*##*/    AuthProcess,
                  L.Connection,
                  L.LineDropped,
                  L.#Retries,
                  L.#AwaitingHooks,
                  L.Msg)))
            &
```

```
              N"PendingRequests = PendingRequests ||
                S"((NewMsgId,
         /*##*/      (Dominates(L.MaxSecLevel,RequestedSecLevel)
                      &
                      Dominates(RequestedSecLevel,L.MinSecLevel) =>
                           WriteAndReadLine
                        <>  ClearLine),
         /*##*/      Undefined,
                     S"((NetworkProcess,
                         nil,
                         NoResponse))))
              &
              (Dominates(L.MaxSecLevel,RequestedSecLevel)
               &
               Dominates(RequestedSecLevel,L.MinSecLevel) =>
                           KernelCalled(SendMessage(NetworkProcess))
                       <>  KernelCalled(SendMessage(NetworkProcess))
                           &
                           KernelCalled(SendMessage(NetworkProcess)))))
```

LGDL2: Network Process message (Retry of LOGON or DIAL)

```
transform LGDL2(Laddr: LineAddress,
                UserId: VirtualMachineName,
                RequestedSecLevel: ProcessName)

refcond E"L:LineEntry(L<:Lines &
                      L.Laddr = Laddr
                      &
                      L.State = AttachValidation
                      &
                      L.CyclePosition = Retry)

effect  E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
            N"Lines = Lines ~~ S"(L) ||
            (Dominates(L.MaxSecLevel,RequestedSecLevel)
             &
             Dominates(RequestedSecLevel,L.MinSecLevel) =>
              S"((L.Laddr,
                  L.MaxSecLevel,
                  L.MinSecLevel,
                  L.State,
      /*##*/      ReadInitialPassword,
      /*##*/      RequestedSecLevel,
      /*##*/      UserId,
                  L.Connection,
                  L.LineDropped,
                  L.#Retries,
                  L.#AwaitingHooks,
                  L.Msg))
             <> S"((L.Laddr,
                  L.MaxSecLevel,
                  L.MinSecLevel,
      /*##*/      Free,
      /*##*/      ReEnablePending,
                  L.RequestedSecLevel,
      /*##*/      AuthProcess,
                  L.Connection,
                  L.LineDropped,
                  L.#Retries,
                  L.#AwaitingHooks,
                  L.Msg)))
            &
```

```
                N"PendingRequests = PendingRequests ||
                  S"((NewMsgId,
          /*##*/      (Dominates(L.MaxSecLevel,RequestedSecLevel)
                       &
                       Dominates(RequestedSecLevel,L.MinSecLevel) =>
                            WriteAndReadLine
                        <>  ClearLine),
          /*##*/      Undefined,
                      S"((NetworkProcess,
                          nil,
                          NoResponse))))
                 &
                 (Dominates(L.MaxSecLevel,RequestedSecLevel)
                  &
                  Dominates(RequestedSecLevel,L.MinSecLevel) =>
                            KernelCalled(SendMessage(NetworkProcess))
                        <>  KernelCalled(SendMessage(NetworkProcess))
                            &
                            KernelCalled(SendMessage(NetworkProcess))))
```

LGDL3: Userid, password, and requested security level validations

```
transform LGDL3(Laddr: LineAddress,
                Password: String)

refcond E"L:LineEntry(L<:Lines &
                      L.Laddr = Laddr
                      &
                      L.State = AttachValidation
                      &
                      L.CyclePosition = ReadInitialPassword)

effect  E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
             N"Lines = Lines ~~ S"(L) ||
               S"((L.Laddr,
                   L.MaxSecLevel,
                   L.MinSecLevel,
        /*##*/      ((E"D:DirectoryEntry(D<:UserDirectory &
                         D.UserId = L.AttachedVM
                         &
                         (L.Connection = Logon ->
                                D.LogonPassword = Password)
                         &
                         (L.Connection = Dial ->
                                D.DialPassword = Password)
                    &
                    ~(Dominates(D.MaxSecLevel,L.RequestedSecLevel)
                      &
                      Dominates(L.RequestedSecLevel,D.MinSecLevel))))
                    |
                    L.#Retries + 1 = #MaxRetries =>
                         Free
                    <>   L.State),
/*##*/     (E"D:DirectoryEntry(D<:UserDirectory &
                  D.UserId = L.AttachedVM
                  &
                  (L.Connection = Logon ->
                          D.LogonPassword = Password)
                  &
                  (L.Connection = Dial ->
                          D.DialPassword = Password)) =>
                  E"D:DirectoryEntry(D<:UserDirectory &
                    D.Userid = L.AttachedVM
                    &
                  (Dominates(D.MaxSecLevel,L.RequestedSecLevel)
                     &
                     Dominates(L.RequestedSecLevel,D.MinSecLevel) =>
                          (E"A:AccessPasswordEntry(
                             A<:D.AccessPasswords &
                              A.SecLevel = L.RequestedSecLevel) =>
                                  ReadAccessPassword
                             <>   PerformResourceChecks)
                     <>   ReEnablePending))
             <>   (L.#Retries + 1 = #MaxRetries =>
                          ReEnablePending
                     <>   Retry)),
            L.RequestedSecLevel,
/*##*/     ((E"D:DirectoryEntry(D<:UserDirectory &
                  D.UserId = L.AttachedVM
                  &
                  (L.Connection = Logon ->
                          D.LogonPassword = Password)
                  &
                  (L.Connection = Dial ->
                          D.DialPassword = Password)
             &
             ~(Dominates(D.MaxSecLevel,L.RequestedSecLevel)
               &
               Dominates(L.RequestedSecLevel,D.MinSecLevel))))
             |
             L.#Retries + 1 = #MaxRetries =>
                  AuthProcess
             <>   L.AttachedVM),
            L.Connection,
            L.LineDropped,
/*##*/     (~(E"D:DirectoryEntry(D<:UserDirectory &
                  D.UserId = L.AttachedVM
                  &
                  (L.Connection = Logon ->
                          D.LogonPassword = Password)
                  &
                  (L.Connection = Dial ->
                          D.DialPassword = Password))) =>
                  L.#Retries + 1
             <>   L.#Retries),
            L.#AwaitingHooks,
                L.Msg))
            &
```

```
E"Entry1, Entry2:PendingRequest(
  Entry1 = (NewMsgId,
            WriteAndReadLine,
            Undefined,
            S"((NetworkProcess,
                nil,
                NoResponse)))
  &
  Entry2 = (NewMsgId,
            ClearLine,
            Undefined,
            S"((NetworkProcess,
                nil,
                NoResponse)))
&
    N"PendingRequests =
    (E"D:DirectoryEntry(D<:UserDirectory &
             D.UserId = L.AttachedVM
             &
             (L.Connection = Logon ->
                    D.LogonPassword = Password)
             &
             (L.Connection = Dial ->
                    D.DialPassword = Password)) =>
             E"D:DirectoryEntry(D<:UserDirectory &
               D.UserId = L.AttachedVM
               &
               (Dominates(D.MaxSecLevel,L.RequestedSecLevel)
                &
                Dominates(L.RequestedSecLevel,D.MinSecLevel) =>
                      (E"A:AccessPasswordEntry(
                         A<:D.AccessPasswords &
                          A.SecLevel = L.RequestedSecLevel) =>
                              PendingRequests || S"(Entry1)
                         <>   PendingRequests)
               <>   PendingRequests || S"(Entry2)))
        <>   (L.#Retries + 1 = #MaxRetries =>
                    PendingRequests || S"(Entry2)
               <>   PendingRequests || S"(Entry1))))
    &
```

```
                    (E"D:DirectoryEntry(D<:UserDirectory &
                            D.UserId = L.AttachedVM
                            &
                            (L.Connection = Logon ->
                                    D.LogonPassword = Password)
                            &
                            (L.Connection = Dial ->
                                    D.DialPassword = Password)) =>
                            E"D:DirectoryEntry(
                               D<:UserDirectory & D.UserId = L.AttachedVM &
                            (Dominates(D.MaxSecLevel,L.RequestedSecLevel)
                              &
                              Dominates(L.RequestedSecLevel,D.MinSecLevel) =>
                                    (E"A:AccessPasswordEntry(
                                      A<:D.AccessPasswords &
                                        A.SecLevel = L.RequestedSecLevel) =>
                                          KernelCalled(SendMessage(
                                               NetworkProcess)))
                              <>    KernelCalled(SendMessage(
                                          NetworkProcess))
                                    &
                                    KernelCalled(SendMessage(
                                          NetworkProcess))))
                <>     (N"L.#Retries = #MaxRetries =>
                                    KernelCalled(SendMessage(
                                          NetworkProcess))
                                    &
                                    KernelCalled(SendMessage(
                                          NetworkProcess))
                              <>    KernelCalled(SendMessage(
                                          NetworkProcess)))))
```

LGDL4: Perform access password checks

```
transform LGDL4(Laddr: LineAddress,
                 AccessPassword: String)

refcond E"L:LineEntry(L<:Lines &
                       L.Laddr = Laddr
                       &
                       L.State = AttachValidation
                       &
                       L.CyclePosition = ReadAccessPassword)

effect  E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
        (~(E"D:DirectoryEntry(D<:UserDirectory &
                              D.UserId = L.UserId
                              &
                              E"A:AccessPasswordEntry(A<:D.AccessPasswords &
                                  A.SecLevel = L.RequestedSecLevel)))
        => Error <> NoError &
           E"D:DirectoryEntry(D<:UserDirectory & D.UserId = L.UserId &
           E"A:AccessPasswordEntry(
             A<:D.AccessPasswords & A.SecLevel = L.RequestedSecLevel &
           N"Lines = Lines ~~ S"(L) ||
             S"((L.Laddr,
                 L.MaxSecLevel,
                 L.MinSecLevel,
        /*##*/   (A.Password ~= AccessPassword
                  &
                  L.#Retries + 1 = #MaxRetries =>
                       Free
                   <>  L.State),
        /*##*/   (A.Password = AccessPassword =>
                       PerformResourceChecks
                   <>  (L.#Retries + 1 = #MaxRetries =>
                              ReEnablePending
                          <>  Retry)),
                 L.RequestedSecLevel,
        /*##*/   (A.Password ~= AccessPassword
                  &
                  L.#Retries + 1 = #MaxRetries =>
                       AuthProcess
                   <>  L.AttachedVM),
                 L.Connection,
                 L.LineDropped,
        /*##*/   (A.Password ~= AccessPassword =>
                       L.#Retries + 1
                   <>  L.#Retries),
                 L.#AwaitingHooks,
                 L.Msg))
           &
```

```
                (A.AccessPassword ~= Password =>
                N"PendingRequests = PendingRequests ||
                  S"((NewMsgId,
          /*##*/      (L.#Retries + 1 = #MaxRetries =>
                            ClearLine
                        <>  WriteAndReadLine),
          /*##*/      Undefined,
                      S"((NetworkProcess,
                          nil,
                          NoResponse))))
                 <> N"PendingRequests = PendingRequests)
                &
                    (A.Password ~= AccessPassword ->
                            KernelCalled(SendMessage(OpProcess))
                            &
                            (L.#Retries + 1 = #MaxRetries =>
                                    KernelCalled(SendMessage(
                                            NetworkProcess))
                                    &
                                    KernelCalled(SendMessage(
                                            NetworkProcess))
                                <>  KernelCalled(SendMessage(
                                            NetworkProcess)))))))
```

LGDL5: Perform resource checks

```
transform LGDL5(Laddr: LineAddress)

refcond E"L:LineEntry(L<:Lines &
                L.Laddr = Laddr)

effect  E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
                (~(E":DirectoryEntry(D<:UserDirectory &
                      D.UserId = L.AttachedVM)
                &
                ~ShuttingDown)
        =>  Error  <>  NoError &
            E"D:DirectoryEntry(D<:UserDirectory & D.UserId = L.AttachedVM &
        /* not specified */  true'))
```

LGDL6: Attach Dedicated Devices

```
transform LGDL6(Laddr: LineAddress)

refcond E"L:LineEntry(L<:Lines &
                     L.Laddr = Laddr
                     &
                     L.State = AttachValidation
                     &
                     L.CyclePosition = HookingPeripherals)

effect  E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
                 (~(E"N:NkcpEntry(N<:CurrentNkcps &
                          N.Process = L.AttachedProcess)
                     &
                     E"D:DirectoryEntry(D<:UserDirectory &
                          D.UserId = L.AttachedVM))
        =>  Error  <>  NoError &
            E"N:NkcpEntry(N<:CurrentNkcps & N.Process = L.AttachedProcess &
            E"D:DirectoryEntry(D<:UserDirectory & D.UserId = L.AttachedVM &
        /* not specified */ true))))
```

LGDL7: Perform Links at Logon

```
transform LGDL7(Laddr: LineAddress)

refcond E"L:LineEntry(L<:Lines &
                      L.Laddr = Laddr
                      &
                      L.State = AttachValidation
                      &
                      L.CyclePosition = HookingPeripherals)

effect  E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
                 (~(E"N:NkcpEntry(N<:CurrentNkcps &
                        N.Process = L.AttachedProcess)
                      &
                      E"D:DirectoryEntry(D<:UserDirectory &
                            D.UserId = L.AttachedVM))
          => Error  <>  NoError &
             E"N:NkcpEntry(N<:CurrentNkcps & N.Process = L.AttachedProcess &
             E"D:DirectoryEntry(D<:UserDirectory & D.UserId = L.AttachedVM &
          /* not specified */  true))))
```

LGDL8: Response to message to NKCP re new VM

```
transform LGDL8(VM: VirtualMachineName,
                Process: ProcessName,
                Laddr: LineAddress)

refcond E"L:LineEntry(L<:Lines &
                    L.Laddr = Laddr)

effect  E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
                (~(L.AttachedVM = VM
                    &
                    L.State = AttachValidation
                    &
                    L.CyclePosition = NotifyingNkcp)
        =>  Error  <>  NoError &
            N"Lines = Lines ~~ S"(L) ||
              S"((L.Laddr,
                  L.MaxSecLevel,
                  L.MinSecLevel,
        /*##*/    (Responded(Process) =>
                              Attached
                       <>    (A"N:NkcpEntry(N<:CurrentNkcps ->
                                    N.Process ~= Process)
                               &
                               L.LineDropped =>
                                      Free
                                 <>   L.State)),
        /*##*/    (Responded(Process) =>
                              Attached
                       <>    (E"N:NkcpEntry(N<:CurrentNkcps &
                                      N.Process = Process)
                                |
                                L.LineDropped =>
                                      L.CyclePosition
                                 <>   PerformResourceChecks)),
                  L.RequestedSecLevel,
                  L.AttachedVM,
                  L.Connection,
                  L.LineDropped,
                  L.#Retries,
                  L.#AwaitingHooks,
                  L.Msg))
            &
```

```
                  (Responded(Process) =>
                          KernelCalled(SendMessage(NetworkProcess))
                     <>   (E"N:NkcpEntry(N<:CurrentNkcps &
                                   N.Process = Process) =>
                                  KernelCalled(SendMessage(Process))
                              <>    (L.LineDropped ->
                                           KernelCalled(SendMessage(
                                                   NetworkProcess)))))))
```

NKCP1: Disconnect

```
transform NKCP1(Process: ProcessName,
                VM: VirtualMachineName,
                Laddr: LineAddress
                LineAction: String)

effect  (~(E"N:NkcpEntry(N<:CurrentNkcps &
                     N.Process = Process)
                &
                E"L:LineEntry(L<:Lines &
                     L.Laddr = Laddr))
        => Error  <>  NoError &
        E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
            &
            E"N:NkcpEntry(N<:CurrentNkcps & N.Process = Process &
            N"Lines = Lines ~~ S"(L) ||
              S"((L.Laddr,
                  L.MaxSecLevel,
                  L.MinSecLevel,
        /*##*/     (E"V:VMEntry(V<:N.VMs &
                                V.VMName = VM
                                 &
                                 V.Laddr = Laddr
                                 &
                                 ~V.Disconnected) =>
                                 Free
                            <>   L.State),
        /*##*/     (E"V:VMEntry(V<:N.VMs &
                                V.VMName = VM
                                 &
                                 V.Laddr = Laddr
                                 &
                                 ~V.Disconnected) =>
                                 ReEnablePending
                            <>   L.CyclePosition),
                  L.RequestedSecLevel.
                  L.AttachedVM,
                  L.Connection,
                  L.LineDropped,
                  L.#Retries,
                  L.#AwaitingHooks,
                  L.Msg))
                &
N"#Users =
        (E"V:VMEntry(V<:N.VMs &
               V.VMName = VM
                &
                V.Laddr = Laddr
                &
                ~V.Disconnected) =>
                #Users - 1
           <>   #Users)
&
N"PendingRequests =
        (E"V:VMEntry(V<:N.VMs &
               V.VMName = VM
                &
                V.Laddr = Laddr
                &
                ~V.Disconnected) =>
                PendingRequests ||
                  S"((NewMsgId,
                      (LineAction = 'hold' =>
                                     ReDirectLine
                         <>          ClearLine),
                      Undefined,
                      S"((NetworkProcess,
                          nil,
                          NoResponse))))
           <>   PendingRequests)
&
(E"V:VMEntry(V<:N.VMs &
        V.VMName = VM
         &
         V.Laddr = Laddr
         &
         ~V.Disconnected) ->
               E"V:VMEntry(V<:N.VMs & V.VMName = VM &
                 V.Laddr = Laddr
                 &
                 (~V.Disconnected)
                 &
                 N"CurrentNkcps = CurrentNkcps ~~ S"(N) ||
                   S"((N.Process,
                       N.VMs ~~ S"(V) ||
                         S"((V.VMName,
                             V.Laddr,
                             true,
                             V.Users)),
                       N.AttachedDevices,
                       N.Links))))
&
KernelCalled(SendMessage(NetworkProcess)))))
```

NKCP2: Logoff

```
transform NKCP2(Process: ProcessName,
                VM: VirtualMachineName,
                LineAction: String,
                ReasonForLogoff: LogoffReasons)

effect  (~(E"N:NkcpEntry(N<:CurrentNkcps &
                    N.Process = Process
                    &
                    E"V:VMEntry(V<:N.VMs &
                          V.VMName = VM)))
        =>  Error  <>  NoError &
            E"N:NkcpEntry(N<:CurrentNkcps & N.Process = Process &
            E"V:VMEntry(V<:N.VMs & V.VMName = VM &
                N"#Users =
                        (DestroyedVM =>
                                #Users - C"V.Users - 1
                           <>   #Users)
                &
                N"#VMs =
                        (DestroyedVM =>
                                #VMs - 1
                           <>   #VMs)
                &
                N"CurrentNkcps =
                        (DestroyedVM =>
                                CurrentNkcps ~~ S"(N) ||
                                S"((N.Process,
                                    N.VMs ~~ S"(V),
                                    N.AttachedDevices,
                                    N.Links))
                         <>     CurrentNkcps)
                &
                N"PendingRequests =
                        (DestroyedVM
                           &
                           ~ V.Disconnected =>
                                PendingRequests || S"P:PendingRequest(
                                   E"U:LineAddress(U<:V.Users &
                                     P=(NewMsgId,
                                        ClearLine(U),
                                        Undefined,
                                        S"((NetworkProcess,
                                            nil,
                                            NoResponse)))))
                                ||
                                        S"((NewMsgId,
                                            (LineAction = 'hold' =>
                                                ReDirectLine
                                            <>  ClearLine),
                                            Undefined,
                                            S"((NetworkProcess,
                                                nil,
                                                NoResponse))))
                                  <>    PendingRequests)
                &
            N"Lines = Lines
                --
                S"L:LineEntry(L<:Lines &
                E"U:LineAddress(U<:V.Users &
                  U = L.Laddr))
                ||
                S"L1:LineEntry(E"L:LineEntry(L<:Lines &
                      E"U:LineAddress(U<:V.Users & U = L.Laddr &
                      L1 = (L.Laddr,
                            L.MaxSecLevel,
                            Free,
                            ReEnablePending,
                            L.RequestedSecLevel,
                            L.AttachedVM,
                            L.Connection,
                            L.LineDropped,
                            L.#Retries,
                            L.#AwaitingHooks,
                            L.Msg))))
                &
                KernelCalled(DestroyVM(V.VMName))
                &
                (DestroyedVM =>
                        (V.Disconnected =>
                                        KernelCalled(SendMessage(OpProcess))
                                  <>    KernelCalled(SendMessage(OpProcess))
                                        &
                                        KernelCalled(SendMessage(
                                                NetworkProcess))
                                        &
                                        KernelCalled(SendMessage(
                                                NetworkProcess))
                                        &
                                        KernelCalled(SendMessage(
                                                AcntProcess))
                                        &
                                        A"U:LineAddress(U<:V.Users ->
                                            KernelCalled(SendMessage(
                                                NetworkProcess))
                                            &
                                            KernelCalled(SendMessage(
                                                NetworkProcess))))
                    <>  KernelCalled(SendMessage(N.Process))))))
```

OP1: Autolog

```
transform OP1(UserId: VirtualMachineName,
                 RequestedSecLevel: ProcessName,
                 Password: String,
                 AccessPassword: String)

effect  (~(E"D:DirectoryEntry(D<:UserDirectory &
                     D.UserId = UserId
                     &
                     D.LogonPassword = Password
                     &
                     Dominates(D.MaxSecLevel,RequestedSecLevel)
                     &
                     Dominates(RequestedSecLevel,D.MinSecLevel)
                     &
                     D.IplDefined = true
                     &
                     A"A:AccessPasswordEntry(A<:D.AccessPasswords &
                         A.SecLevel = RequestedSecLevel ->
                                A.Password = AccessPassword))
                 &
                 ~ ShuttingDown
                 &
                 #VMs < #MaxVMs)
         => Error <> NoError &
         /* not specified */ true)
```

UR2 and OP2: Map user id

```
transform UR2(UserId: VirtualMachineName,
              Requester: ProcessName)

refcond Requester <: S"(OpProcess, URProcess)

effect  KernelCalled(SendMessage(Requester))
```

OP4b: Detach of shared device (by operator)

```
transform OP4b(Raddr: DeviceAddress)

refcond E"S:SharableDriveEntry(S<:SharableDrives &
                    S.Raddr = Raddr)

effect  E"S:SharableDriveEntry(
              S<:SharableDrives & S.Raddr = Raddr &
            N"SharableDrives = SharableDrives ~~ S"(S) ||
              S"((S.Raddr,
        /*##*/     (S.State = AttachedToSystem
                    &
                    A"M:MiniDiskEntry(M<:MiniDisks &
                        M.ContainingVolume = S.MountedVolume ->
                                M.CurrentLinks = Empty)
                    &
                    DeviceIsReleasable
                    &
                    DeviceReleased =>
                        Available
                    <>  S.State),
                 S.SecLevel,
                 S.MountedVolume))
               &
               (S.State = AttachedToSystem
                &
                A"M:MiniDiskEntry(M<:MiniDisks &
                    M.ContainingVolume = S.MountedVolume ->
                        M.CurrentLinks = Empty)
                &
                DeviceIsReleasable
                &
                DeviceReleased ->
                        E"V:SharedVolumeEntry(V<:SharedVolumes &
                              V.MountedDevice = S.MountedVolume
                              &
               N"SharedVolumes = SharedVolumes ~~ S"(V) ||
                 S"((V.Volume,
                      V.SecLevel,
                      nil,
                      NotMounted))))
                    &
                    KernelCalled(SendMessage(OpProcess))
                    &
                    (S.State = AttachedToSystem ->
                            (A"M:MiniDiskEntry(M<:MiniDisks &
                                    M.ContainingVolume = S.MountedVolume ->
                                            M.CurrentLinks = Empty) ->
                                     KernelCalled(IsDeviceReleasable)
                                     &
                                     (DeviceIsReleasable ->
                                            KernelCalled(ReleaseDevice)))))
```

OP4a: Detach of nonshared device (by operator)

```
transform OP4a(Raddr: DeviceAddress)

refcond E"NS:NonsharableDriveEntry(NS<:NonsharableDrives &
                    NS.Raddr = Raddr)

effect  E"NS:NonsharableDriveEntry(
             NS<:NonsharableDrives & NS.Raddr = Raddr &
           N"NonsharableDrives = NonsharableDrives ~~ S"(NS) ||
             S"((NS.Raddr,
                 NS.MaxSecLevel,
                 NS.MinSecLevel,
        /*##*/    (NS.State = AttachedToUser =>
                        DetachPending
                    <>  NS.State),
                 NS.AttachedProcess,
                 NS.Access))
               &
               N"PendingRequests =
                       (NS.State = AttachedToUser =>
                               PendingRequests ||
                               S"((NewMsgId,
                                   RelinquishDevice,
                                   DETACH+RADDR,
                                   S"((NetworkProcess,
                                       nil,
                                       NoResponse))))
                           <>  PendingRequests)
               &
               (NS.State = AttachedToUser =>
                       KernelCall(SendMessage(NS.AttachedProcess))
                  <>   (NS.State <: S"(OffLine, Available) ->
                               KernelCalled(SendMessage(OpProcess)))))
```

OP5a: Vary (both online and offline) of shared device

```
transform OP5a(Raddr: DeviceAddress,
               Parameter: String)

refcond E"S:SharableDriveEntry(S<:SharableDrives &
                   S.Raddr = Raddr)

effect  (Parameter ~<: S"('online, 'offline')
        => Error  <>  NoError &
           E"S:SharableDriveEntry(
             S<:SharableDrives & S.Raddr = Raddr &
           N"SharableDrives = SharableDrives ~~ S"(S) ||
             S"((S.Raddr,
        /*##*/      (Parameter = 'online' =>
                          Available
                      <>  OffLine),
                   S.SecLevel,
                   S.MountedVolume))
               &
               KernelCalled(SendMessage(OpProcess))))
```

OP5b: Vary (both online and offline) of nonshared device

```
transform OP5b(Raddr: DeviceAddress,
               Parameter: String)

refcond E"NS:NonsharableDriveEntry(NS<:NonsharableDrives &
                  NS.Raddr = Raddr)

effect  (Parameter ~<: S"('online', 'offline')
        => Error  <>  NoError &
           E"NS:NonsharableDriveEntry(
             NS<:NonsharableDrives & NS.Raddr = Raddr &
           N"NonsharableDrives = NonsharableDrives ~~ S"(NS) ||
             S"((NS.Raddr,
                 NS.MaxSecLevel,
                 NS.MinSecLevel,
        /*##*/    (Parameter = 'online' =>
                       Available
                    <>  OffLine),
                 NS.AttachedProcess,
                 NS.Access))
              &
              KernelCalled(SendMessage(OpProcess))))
```

```
OP6a: QUERY, with parameters:
                DASD
                LINES
                GRAF
                ALL
                NAMES
                USERS with no further parameter

transform OP6a(Command:CommandName)

refcond Command<:Cat6a

effect  KernelCalled(SendMessage(OpProcess))
```

```
OP6b: QUERY, with parameters:
                raddr
                SYSTEM raddr


transform OP6b(Command: CommandName,
                Raddr: DeviceAddress)

refcond Command<:Cat6b

effect   (~(E"S:SharableDriveEntry(S<:SharableDrives &
                    S.Raddr = Raddr)
                |
                E"NS:NonsharableDriveEntry(NS<:NonsharableDrives &
                    NS.Raddr = Raddr))
         => Error <> NoError &
         KernelCalled(SendMessage(OpProcess)))
```

```
OP6c: QUERY, with parameters:
                 USERS userid
                 userid

transform OP6c(Command: CommandName,
                 UserId: VirtualMachineName)

refcond Command<:Cat6c

effect  KernelCalled(SendMessage(OpProcess))
```

OP7: LOCATE←RADDR

```
transform OP7(Raddr: DeviceAddress)

refcond E"NS:NonsharableDriveEntry(NS<:NonsharableDrives &
                        NS.Raddr = Raddr)

effect  E"NS:NonsharableDriveEntry(
                     NS<:NonsharableDrives & NS.Raddr = Raddr &
              N"PendingRequests =
              (NS.State = AttachedToUser =>
              PendingRequests ||
                S"((NewMsgId,
                     OpRequest,
                     LOCATE←RADDR,
         /*##*/      S"((NS.AttachedProcess,
                         nil,
                         NoResponse))))
                               <>   PendingRequests)
                   &
                   (NS.State = AttachedToUser =>
                           KernelCalled(SendMessage(NS.AttachedProcess))
                      <>   KernelCalled(SendMessage(OpProcess))))
```

OP8: Shutdown

```
transform OP8

effect  N"ShuttingDown = true
```

OP3: Attach (nonsharable disk drive) Device

```
transform OP3(Raddr: DeviceAddress,
              Process: ProcessName,
              VolSecLevel: ProcessName,
              Access: set of AccessModes)

refcond E"NS:NonsharableDriveEntry(NS<:NonsharableDrives &
                    NS.Raddr = Raddr)

effect  E"NS:NonsharableDriveEntry(
             NS<:NonsharableDrives & NS.Raddr = Raddr &
              (~(E"N:NkcpEntry(N<:CurrentNkcps &
                  N.Process = Process
                  &
                  A"A:AttachedDeviceEntry(A<:N.AttachedDevices ->
                      A.Raddr ~= Raddr)
                  &
                  NS.State = Free
                  &
                  Dominates(N.Process,VolSecLevel)
                  &
                  Dominates(NS.MaxSecLevel,N.Process)
                  &
                  Dominates(N.Process,NS.MinSecLevel)
                  &
                  Dominates(NS.MaxSecLevel,VolSecLevel)
                  &
                  Dominates(VolSecLevel,NS.MinSecLevel))
              &
              Access ~= Empty
              &
              ~ ShuttingDown)
        => Error <> NoError &
```

```
    E"N:NkcpEntry(N<:CurrentNkcps & N.Process = Process &
    N"NonsharableDrives = NonsharableDrives ~~ S"(NS) ||
      S"((NS.Raddr,
          NS.MaxSecLevel,
          NS.MinSecLevel,
/*##*/     ((Write<:Access & VolSecLevel ~= N.Process =>
/*##*/          Access ~~ S"(Write) <> Access) ~= Empty
                   &
                   GrantedAccess =>
                        Attached
                   <>   NS.State),
/*##*/     ((Write<:Access & VolSecLevel ~= N.Process =>
/*##*/          Access ~~ S"(Write) <> Access) ~= Empty
                   &
                   GrantedAccess =>
                        N.Process
                   <>   NS.AttachedProcess),
/*##*/     (Write <: Access
                   &
                   VolSecLevel ~= N.Process =>
                        Access ~~ S"(Write)
                   <>   Access)))
        &
N"CurrentNkcps = CurrentNkcps ~~ S"(N) ||
      S"((N.Process,
          N.VMs,
/*##*/     ((Write<:Access & VolSecLevel ~= N.Process =>
/*##*/          Access ~~ S"(Write) <> Access) ~= Empty
            &
            GrantedAccess =>
                NS.AttachedDevices ||
                S"((NS.Raddr,
                    NS.Access))
            <>  N.AttachedDevices),
          N.Links))
        &
        ((Write<:Access & VolSecLevel ~= N.Process =>
                Access ~~ S"(Write)
             <> Access)
         = Empty =>
                KernelCalled(SendMessage(OpProcess))
           <>   KernelCalled(GrantAccess)
                &
                (GrantedAccess =>
                        KernelCalled(SendMessage(N.Process))
                        &
                        KernelCalled(SendMessage(OpProcess))
                   <>   KernelCalled(SendMessage(OpProcess))))))
```

UR1: URProcess request: need Nkcp

```
transform UR1(RequestedSecLevel: ProcessName,
                Raddr: DeviceAddress)

effect   (A"D:URPOwnedDeviceEntry(D<:URPOwnedDevices ->
                  D.Raddr ~= Raddr)
         => Error  <>  NoError &
                E"D:URPOwnedDeviceEntry(
                  D<:URPOwnedDecices & D.Raddr = Raddr &
                N"#Nkcps =
                        (A"N:NkcpEntry(N<:CurrentNkcps ->
                              N.Process ~= RequestedSecLevel)
                          &
                          Dominates(D.MaxSecLevel,RequestedSecLevel)
                          &
                          Dominates(RequestedSecLevel,D.MinSecLevel)
                          &
                          #Nkcps < #MaxNkcps
                          &
                          CreatedProcess =>
                               #Nkcps + 1
                          <>   #Nkcps)
                &
                N"CurrentNkcps =
                        (A"N:NkcpEntry(N<:CurrentNkcps ->
                              N.Process ~= RequestedSecLevel)
                          &
                          Dominates(D.MaxSecLevel,RequestedSecLevel)
                          &
                          Dominates(RequestedSecLevel,D.MinSecLevel)
                          &
                          #Nkcps < #MaxNkcps
                          &
                          CreatedProcess =>
                               CurrentNkcps ||
                               S"((RequestedSecLevel,
                                   Empty,
                                   Empty,
                                   Empty))
                          <>   CurrentNkcps)
                &
```

```
                    (E"N:NkcpEntry(N<:CurrentNkcps &
                         N.Process = RequestedSecLevel) =>
                            KernelCalled(SendMessage(URProcess))
                      <>    (Dominates(D.MaxSecLevel,RequestedSecLevel)
                               &
                               Dominates(RequestedSecLevel,D.MinSecLevel)
                               &
                               #Nkcps < #MaxNkcps =>
                                    KernelCalled(CreateProcess)
                                    &
                                    (CreatedProcess =>
                                           KernelCalled(SendMessage(
                                                   URProcess))
                                           &
                                           KernelCalled(SendMessage(
                                                   OpProcess))
                                           &
                                           KernelCalled(SendMessage(
                                                   URProcess))
                                       <>  KernelCalled(SendMessage(
                                                   URProcess)))
                               <>   KernelCalled(SendMessage(URProcess)))))
```

UR3a: URProcess response to device attachment request (attach succeeded)

```
transform UR3a(Raddr: DeviceAddress,
               Process: ProcessName,
               Laddr: LineAddress)

effect  (~(E"A:URPOwnedDeviceEntry(A<:URPOwnedDevices &
                  A.Raddr = Raddr)
               &
               E"N:NkcpEntry(N<:CurrentNkcps &
                  N.Process = Process)
               &
               E"L:LineEntry(L<:Lines &
                  L.Laddr = Laddr
                  &
                  L.State = AttachValidation
                  &
                  L.CyclePosition = HookingPeripherals
                  &
                  L.#AwaitingHooks > 0))
        => Error <> NoError &
        E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
            N"Lines = Lines ~~ S"(L) ||
              S"((L.Laddr,
                  L.MaxSecLevel,
                  L.MinSecLevel,
                  L.State,
        /*##*/     (L.#AwaitingHooks - 1 = 0 =>
                        NotifyingNkcp,
                    <> L.CyclePosition),
                  L.RequestedSecLevel,
                  L.AttachedVM,
                  L.Connection,
                  L.LineDropped,
                  L.#Retries,
        /*##*/    L.#AwaitingHooks - 1,
        /*##*/    Concat(L.Msg,Avail(Raddr))))
            &
```

```
                N"PendingRequests =
                        (L.#AwaitingHooks - 1 = 0 =>
                                PendingRequests ||
                                  S"((NewMsgId,
                                      NewVM,
                                      Undefined,
                                      S"((Process,
                                          nil,
                                          NoResponse))))
                           <>   PendingRequests)
                &
                (L.#AwaitingHooks - 1 = 0 ->
                        (~L.LineDropped ->
                                KernelCalled(SendMessage(
                                        NetworkProcess)))
                        &
                       KernelCalled(SendMessage(Process))))
```

UR3b: URProcess response to device attachment request (attach failed)

```
transform UR3b(Raddr: DeviceAddress,
               Process: ProcessName,
               Laddr: LineAddress)

effect  (~(E"A:URPOwnedDeviceEntry(A<:URPOwnedDevices &
                A.Raddr = Raddr)
            &
            E"N:NkcpEntry(N<:CurrentNkcps &
                N.Process = Process)
            &
            E"L:LineEntry(L<:Lines &
                L.Laddr = Laddr
                &
                L.State = AttachValidation
                &
                L.CyclePosition = HookingPeripherals
                &
                L.#AwaitingHooks > 0))
        => Error  <>  NoError &
        E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
            N"Lines = Lines ~~ S"(L) ||
              S"((L.Laddr,
                  L.MaxSecLevel,
                  L.MinSecLevel,
                  L.State,
        /*##*/    (L.#AwaitingHooks - 1 = 0 =>
                                NotifyingNkcp
                           <>   L.CyclePosition),
                  L.RequestedSecLevel,
                  L.AttachedVM,
                  L.Connection,
                  L.LineDropped,
                  L.#Retries,
        /*##*/    L.#AwaitingHooks - 1,
        /*##*/    Concat(L.Msg,Unavail(Raddr))))
            &
N"PendingRequests =
        (L.#AwaitingHooks - 1 = 0 =>
                PendingRequests ||
                        S"((NewMsgId,
                            NewVM,
                            Undefined,
                            S"((Process,
                                nil,
                                NoResponse))))
          <>    PendingRequests)
&
(L.#AwaitingHooks = 1 - 0 ->
        (~L.LineDropped ->
                KernelCalled(SendMessage(
                        NetworkProcess)))
        &
        KernelCalled(SendMessage(Process)))))
```

NKCP3: Drop User

```
transform NKCP3(Process: ProcessName,
                VM: VirtualMachineName,
                Laddr: LineAddress)

effect   (~(E"N:NkcpEntry(N<:CurrentNkcps &
                    N.Process = Process
                    &
                    E"V:VMEntry(V<:N.VMs &
                        V.VMName = VM
                        &
                        E"U:LineAddress(U<:V.Users &
                            U = Laddr)))
                &
                E"L:LineEntry(L<:Lines &
                    L.Laddr = Laddr
                    &
                    L.State = Attached
                    &
                    L.Connection = Dial
                    &
                    L.AttachedVM = VM))
         => Error  <>  NoError &
         E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
             N"Lines = Lines ~~ S"(L) ||
               S"((L.Laddr,
                   L.MaxSecLevel,
                   L.MinSecLevel,
         /*##*/    Free,
         /*##*/    ReEnablePending,
                   L.RequestedSecLevel,
                   L.AttachedVM,
                   L.Connection,
                   L.LineDropped,
                   L.#Retries,
                   L.#AwaitingHooks,
                   L.Msg))
             &
E"N:NkcpEntry(N<:CurrentNkcps & N.Process = Process &
E"V:VMEntry(V<:N.VMs& V.VMName = VM &
N"CurrentNkcps = CurrentNkcps ~~ S"(N) ||
  S"((N.Process,
       N.VMs ~~ S"(V) ||
         S"((V.VMName,
              V.Laddr,
              V.Disconnected,
              V.Users ~~ S"(Laddr))),
       N.AttachedDevices,
       N.Links))))
    &
    N"PendingRequests = PendingRequests ||
       S"((NewMsgId,
            ClearLine,
            Undefined,
            S"((NetworkProcess,
                 nil,
                 NoResponse))))
    &
    KernelCalled(SendMessage(NetworkProcess))
    &
    KernelCalled(SendMessage(OpProcess))
    &
    KernelCalled(SendMessage(AcntProcess))))
```

NTWK3: Link password received

```
transform NTWK3(Process: ProcessName,
                Password: String,
                Requester: VirtualMachineName,
                Laddr: LineAddress,
                User: VirtualMachineName,
                MiniDisk: MiniDiskName,
                RequestedAccess: LinkAccess)

refcond E"L:LineEntry(L<:Lines &
                      L.Laddr = Laddr
                      &
                      L.State = Attached
                      &
                      L.CyclePosition = ReadLinkPassword)

effect  (~(E"N:NkcpEntry(N<:CurrentNkcps &
                   N.Process = Process))
        => Error  <>  NoError &
           E"N:NkcpEntry(N<:CurrentNkcps & N.Process = Process &
           E"D:DirectoryEntry(D<:UserDirectory & D.Userid = User &
        E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
           N"Lines = Lines ~~ S"(L) ||
             S"((L.Laddr,
                 L.MaxSecLevel,
                 L.MinSecLevel,
                 L.State,
       /*##*/    Attached,
                 L.RequestedSecLevel,
                 L.AttachedVM,
                 L.Connection,
                 L.LineDropped,
                 L.#Retries,
                 L.#AwaitingHooks,
                 L.Msg))
          &
          (D.LinkPassword = Password ->
             E"M:MiniDiskEntry(M<:MiniDisks &
               M.MDName = MiniDisk
               &
             E"V:SharedVolumeEntry(V<:SharedVolumes &
               V.Volume = M.ContainingVolume
               &
               (V.State = Mounted ->
                 E"S:SharableDriveEntry(S<:SharableDrives &
                   S.Raddr = V.MountedDevice
                   &
                   (S.State = AttachedToSystem ->
                   E"A:ACLEntry(A<:M.AccessControlList &
                     A.User = Requester
                     &
(RequestedAccess = R =>
    (A"C:ProcessLinkEntry(C<:M.CurrentLinks ->
            Write ~<: C.Access) =>
             Link(N.Process,M.MDName,S"(Read))
       <>    NoLink(PreviousWriteLink))

<> RequestedAccess = RR =>
             Link(N.Process,M.MDName,S"(Read))

<> RequestedAccess = W =>
    (Write <: A.Access =>
             (M.CurrentLinks = Empty =>
                      Link(N.Process,M.MDName,S"(Write))
                 <>   NoLink(PreviousLink))
       <>    NoLink(NoWritePermission))

<> RequestedAccess = WR =>
    (Write <: A.Access =>
             (M.CurrentLinks = Empty =>
                      Link(N.Process,M.MDName,S"(Write))
                 <>   Link(N.Process,M.MDName,S"(Read)))
       <>    /* choices:      Link(N.Process,M.MDName,S"(Read))
                              NoLink(NoWritePermission)   */
             NoLink(NoWritePermission))

<> RequestedAccess = M =>
    (Write <: A.Access =>
             (A"C:ProcessLinkEntry(C<:M.CurrentLinks ->
                     Write ~<: C.Access) =>
                      Link(N.Process,M.MDName,S"(Write))
                 <>   NoLink(PreviousWriteLink))
       <>    NoLink(NoWritePermission))

<> RequestedAccess = MR =>
    (Write <: A.Access =>
             (A"C:ProcessLinkEntry(C<:M.CurrentLinks ->
                     Write ~<: C.Access) =>
                      Link(N.Process,M.MDName,S"(Write))
                 <>   Link(N.Procss,M.MDName,S"(Read)))
       <>    /* choices:      Link(N.Process,M.MDName,S"(Read))
                              NoLink(NoWritePermission)   */
             NoLink(NoWritePermission))

<> RequestedAccess = MW =>
    (Write <: A.Access =>
             Link(N.Process,M.MDName,S"(Write))
       <>    NoLink(NoWritePermission))))))))))))))
```

```
transform Link(Process: ProcessName,
               MiniDisk: MiniDiskName,
               Access:set of AccessModes)

effect  E"N:NkcpEntry(N<:CurrentNkcps & N.Process = Process &
            E"M:MiniDiskEntry(M<:MiniDisks & M.MDName = MiniDisk &
             (E"MPL:ProcessLinkEntry(
                       MPL<:M.CurrentLinks & MPL.Process = N.Process) =>
                  E"MPL:ProcessLinkEntry(
                    MPL<:M.CurrentLinks & MPL.Process = N.Process &
                  E"NML:MDLinkEntry(NML<:N.Links & NML.MDName = M.MDName &
                    (Access ~<<= MPL.Access ->
                         KernelCalled(GrantAccess)
                         &
                         (GrantedAccess =>
                            N"CurrentNkcps = CurrentNkcps ~~ S"(N) ||
                              S"((N.Process,
                                  N.VMs,
                                  N.AttachedDevices,
                                  N.Links ~~ S"(NML) ||
                                    S"((NML.MDName,
                                        Access))))
                            &
                            N"MiniDisks = MiniDisks ~~ S"(M) ||
                              S"((M.MDName,
                                  M.ContainingVolume,
                                  M.Cylinders,
                                  M.SecLevel,
                                  M.CurrentLinks ~~ S"(MPL) ||
                                    S"((MDL.Process,
                                        Access)),
                                  M.AccessControlList))
                         <> N"CurrentNkcps = CurrentNkcps
                            &
                            N"MiniDisks = MiniDisks))')
               <> KernelCalled(GrantAccess)
                  &
```

```
                    (GrantedAccess =>
                         N"CurrentNkcps = CurrentNkcps ~~ S"(N) ||
                           S"((N.Process,
                               N.VMs,
                               N.AttachedDevices,
                               N.Links ||
                                 S"((M.MDName,
                                     Access))))
                         &
                         N"MiniDisks = MiniDisks ~~ S"(M) ||
                           S"((M.MDName,
                               M.ContainingVolume,
                               M.Cylinders,
                               M.SecLevel,
                               M.CurrentLinks ||
                                 S"((N.Process,
                                     Access)),
                               M.AccessControlList))
                     <> N"CurrentNkcps = CurrentNkcps
                        &
                        N"MiniDisks = MiniDisks))))
```

NKCP4: Link (with password)

```
transform NKCP4(Process: ProcessName,
                Requester: VirtualMachineName,
                Laddr: LineAddress,
                User: VirtualMachineName,
                MiniDisk: MiniDiskName,
                RequestedAccess: LinkAccess)

effect  (ShuttingDown
                |
                RequestedAccess = Empty
       =>  Error  <>  NoError &
                (E"N:NkcpEntry(N<:CurrentNkcps &
                        N.Process = Process
                        &
                        E"E:VMEntry(E<:N.VMs &
                             E.VMName = Requester))
                  &
                  E"L:LineEntry(L<:Lines &
                        L.Laddr = Laddr
                        &
                        L.State = Attached
                        &
                        L.CyclePosition = Attached
                        &
                        L.RequestedSecLevel = Process
                        &
                        L.AttachedVM = Requester)
                 &
                 E"D:DirectoryEntry(D<:UserDirectory &
                        D.UserId = User
                        &
                        E"K:MDLinkEntry(K<:D.Links &
                             K.MDName = MiniDisk))
                 &
                 E"M:MiniDiskEntry(M<:MiniDisks &
                        M.MDName = MiniDisk
                        &
                        E"A:ACLEntry(A<:M.AccessControlList &
                        A.User = Requester
                        &
                        Dominates(Process,M.SecLevel)
                        &
                        (Write<:RequestedAccess ->
                             Process = M.SecLevel))
                &
```

```
                                E"V:SharedVolumeEntry(V<:SharedVolumes &
                                    V.Volume = M.ContainingVolume
                                    &
                                    V.State = Mounted
                                    &
                                    E"S:SharableDriveEntry(S<:SharableDrives &
                                        S.Raddr = V.MountedDevice
                                        &
                                        S.State = AttachedToSystem))) =>
                                    KernelCalled(SendMessage(NetworkProcess))
             &
                                    KernelCalled(SendMessage(NetworkProcess))
             &
         E"L:LineEntry(L<:Lines & L.Laddr = Laddr &
             N"Lines = Lines ~~ S"(L) ||
               S"((L.Laddr,
                   L.MaxSecLevel,
                   L.MinSecLevel,
                   L.State,
                   L.CyclePosition,
                   L.RequestedSecLevel,
                   L.AttachedVM,
                   L.Connection,
                   L.LineDropped,
                   L.#Retries,
                   L.#AwaitingHooks,
                   L.Msg))
             &
             N"PendingRequests = PendingRequests ||
               S"((NewMsgId,
                   WriteAndReadLine,
                   Undefined,
                   S"((NetworkProcess,
                       nil,
                       NoResponse))))
         <> N"Lines = Lines
            &
            N"PendingRequests = PendingRequests
            &
            KernelCalled(SendMessage(Process))))
```

NKCPS, NKCPG: Detach nonsharable device (request from process), and response (from process) to relinquish device request from Authprocess

```
transform NKCPS(Raddr: DeviceAddress,
                Process: ProcessName,
                User: VirtualMachineName)

        (~(E"N:NkcpEntry(N<:CurrentNkcps &
                    N.Process = Process
                    &
                    E"VM:VMEntry(VM<:N.VMs &
                         VM.VMName = User)
                    &
                    E"A:AttachedDeviceEntry(A<:N.AttachedDevices &
                         A.Raddr = Raddr))
                 &
                 E"NS:NonsharableDriveEntry(NS<:NonsharableDrives &
                      NS.Raddr = Raddr
                      &
                      NS.AttachedProcess = Process))
        => Error  <>  NoError &
           E"NS:NonsharableDriveEntry, N:NkcpEntry(
             NS<:NonsharableDrives & NS.Raddr = Raddr &
             N<:CurrentNkcps & N.Process = Process &
             N"NonsharableDrives = NonsharableDrives ~~ S"(NS) ||
             S"((NS.Raddr,
                 NS.MaxSecLevel,
                 NS.MinSecLevel,
                 (DeviceReleased =>
                       Available
                   <>  NS.State),
                 NS.AttachedProcess,
                 NS.Access))
           &
           N"CurrentNkcps = CurrentNkcps ~~ S"(N) ||
             S"((N.Process,
                 N.VMs,
                 (DeviceReleased =>
                       N.AttachedDevices --
                               S"A:AttachedDeviceEntry(
                               A<:N.AttachedDevices
                               &
                               A.Raddr = NS.Raddr)
                   <>  N.AttachedDevices),
                 N.Links))
&
KernelCalled(ReleaseDevice(NS.Raddr))
&
KernelCalled(SendMessage(OpProcess))
&
(DeviceReleased =>
         KernelCalled(SendMessage(AcntProcess')
   <>    KernelCalled(SendMessage(NS.AttachedProcess)))))
```

NKCP7: Purge NKCP

```
transform NKCP7(Process: ProcessName)

effect  (~(E"N:NkcpEntry(N<:CurrentNkcps &
                N.Process = Process
                &
                N.VMs = Empty
                &
                N.AttachedDevices = Empty
                &
                N.Links = Empty))
        =>  Error  <>  NoError &
            E"N:NkcpEntry(N<:CurrentNkcps & N.Process = Process &
            N"CurrentNkcps = CurrentNkcps ~~ S"(N)))
```

KERN1: message from Kernel, re shared device availability

```
transform KERN1(Raddr: DeviceAddress,
                Volume: VolumeId,
                CurrentStatus: SharableDriveStatus)

effect  (~(E"S:SharableDriveEntry(S<:SharableDrives &
                    S.Raddr = Raddr
                    &
                    (CurrentStatus = AttachedToSystem ->
                        E"V:SharedVolumeEntry(V<:SharedVolumes &
                            V.Volume = Volume))))
         => Error <> NoError &
            E"S:SharableDriveEntry(S<:SharableDrives & S.Raddr = Raddr &
            (S.State = AttachedToSystem =>
               (CurrentStatus = AttachedToSystem =>
                  (Volume ~= S.MountedVolume =>
                     (E"M:MiniDiskEntry(M<:MiniDisks &
                        M.ContainingVolume = S.MountedVolume
                        &
                        M.CurrentLinks ~= Empty) =>
                           Error
                      <> E"Vold,Vnew:SharedVolumeEntry(
                           Vold<:SharedVolumes & Vnew<:SharedVolumes &
                           Vold.Volume = S.MountedVolume
                           &
                           Vold.State = Mounted
                           &
                           Vold.MountedDevice = S.Raddr
                           &
                           Vnew.Volume = Volume
                           &
                           (Dominates(S.SecLevel,VnewSecLevel) =>
                              N"SharableDrives = SharableDrives ~~ S"(S) ||
                                S"((S.Raddr,
                                    S.State,
                                    S.SecLevel,
                                    Vnew.Volume))
                              &
                              N"SharedVolumes = SharedVolumes
                                ~~ S"(Vold,Vnew)
                                ||
                                S"((Vold.Volume,
                                    Vold.SecLevel,
                                    nil,
                                    NotMounted))
                                ||
                                S"((Vnew.Volume,
                                    Vnew.SecLevel,
                                    S.Raddr,
                                    Mounted))
                              &
                              KernelCalled(DriveMatchesVolume)
                           <> N"SharableDrives = SharableDrives
                              &
                              N"SharedVolumes = SharedVolumes
                              &
                              KernelCalled(DriveDoesNotMatchVolume))))
                  <> N"SharableDrives = SharableDrives
                     &
                     N"SharedVolumes = SharedVolumes
                     &
                     KernelCalled(DriveMatchesVolume))
               <> E"V:SharedVolumeEntry(V<:SharedVolumes &
                    V.Volume = S.MountedVolume
                    &
                    V.State = Mounted
                    &
                    V.MountedDevice = S.Raddr
                    N"SharableDrives = SharableDrives ~~ S"(S) ||
                      S"((S.Raddr,
                          CurrentStatus,
                          S.SecLevel,
                          nil))
                    &
                    N"SharedVolumes = SharedVolumes ~~ S"(V) ||
                      S"((V.Volume,
                          V.SecLevel,
                          nil,
                          NotMounted))))
            <> (CurrentStatus = AttachedToSystem =>
                   E"V:SharedVolumeEntry(V<:SharedVolumes &
                    V.Volume = Volume
                    &
                    (Dominates(S.SecLevel,V.SecLevel) =>
                       N"SharableDrives = SharableDrives ~~ S"(S) ||
                         S"((S.Raddr,
                             AttachedToSystem,
                             S.SecLevel,
                             V.Volume))
                       &
                       N"SharedVolumes = SharedVolumes ~~ S"(V) ||
                         S"((V.Volume,
                             V.SecLevel,
                             S.Raddr,
                             Mounted))
                       &
                       KernelCalled(DriveMatchesVolume)
                    <> N"SharableDrives = SharableDrives
                       &
                       N"SharedVolumes = SharedVolumes
                       &
                       KernelCalled(DriveDoesNotMatchVolume)))
```

```
<> N"SharablcDrives = SharableDrives ~~ S"(S) ||
     S"((S.Raddr,
         CurrentStatus,
         S.SecLevel,
         S.Volume))))))
```

```
transform MsgOp(MsgId: MessageId,
                Text: String,
                Source: ProcessName)

refcond Source = OpProcess

effect  (E"P:PendingRequest(P<:PendingRequests
                           &
                           P.MsgId = MsgId) =>
             Error
        <> (MsgName(Text) = AUTOLOG =>
                OP1
           <> MsgName(Text) = ATTACH←RADDR =>
                (E"NS: NonsharableDriveEntry(NS<:NonsharableDrives &
                   NS.Raddr = Raddr(Text)) =>
                    OP3
                <> Error)
           <> MsgName(Text) = DETACH←RADDR =>
                (E"NS: NonsharableDriveEntry(NS<:NonsharableDrives &
                   NS.Raddr = Raddr(Text)) =>
                    OP4a
                <> E"S: SharableDriveEntry(S<:SharableDrives &
                     S.Raddr = Raddr(Text)) =>
                    OP4b
                <> Error)
           <> MsgName(Text)<:S"(VARY←ONLINE, VARY←OFFLINE) =>
                (E"NS: NonsharableDriveEntry(NS<:NonsharableDrives &
                   NS.Raddr = Raddr(Text)) =>
                    OP5b
                <> E"S: SharableDriveEntry(S<:SharableDrives &
                     S.Raddr = Raddr(Text)) =>
                    OP5a
                <> Error)
           <> MsgName(Text)<:Cat6a =>
                    OP6a
           <> MsgName(Text)<:Cat6b
                    OP6b
           <> MsgName(Text)<:Cat6c =>
                    OP6c
           <> MsgName(Text) = LOCATE←RADDR =>
                    OP7a
           <> MsgName(Text) = SHUTDOWN =>
                    OP8
           <> MsgName(Text) = MapUserId =>
                    OP2
           <> Error))
```

```
transform MsgUR(MsgId: MessageId,
                Text: String,
                Source: ProcessName)

refcond Source = URProcess

effect  (E"P:PendingRequest(P<:PendingRequests &
           P.MsgId = MsgId) =>
           E"P:PendingRequest(P<:PendingRequests &
             P.MsgId = MsgId
             &
             (P.Kind = Attach =>
                (MsgName(Text) = Attached =>
                   UR3a
                <> MsgName(Text)<:S"(AttachFailed,DeviceNotAvailable) =>
                   UR3b
                <> Error)
             <> Error))
        <> (MsgName(Text) = NeedNkcp =>
              UR1
           <> MsgName(Text) = MapUserId =>
              UR2
           <> Error))
```

```
transform MsgNet(MsgId: MessageId,
                 Text: String,
                 Source: ProcessName)

refcond Source = NetworkProcess

effect  (E"P:PendingRequest(P<:PendingRequests &
          P.MsgId = MsgId) =>
          E"P:PendingRequest(P<:PendingRequests &
               P.MsgId = MsgId
               &
               (P.Kind<:S"(ClearLine, ReDirectLine) =>
                  (MsgName(Text) = LineStatus =>
                     NTWK1
                  <> Error)
               <> P.Kind = WriteAndReadLine =>
                     (MsgName(Text) = LineInfo =>
                       (E"L:LineEntry(L<:Lines &
                          L.Laddr = Laddr(Text)) =>
                          E"L:LineEntry(L<:Lines &
                            L.Laddr = Laddr(Text)
                            &
                            (L.State = Attached =>
                                (L.CyclePosition = ReadLinkPassword =>
                                   NTWK3
                                <> Error)
                            <> L.State = AttachValidation =>
                                  (L.CyclePosition = Retry =>
                                     LGDL2
                                  <> L.CyclePosition = ReadInitialPassword =>
                                        LGDL3
                                        &
                                        (TEMP"L.CyclePosition =
                                                PerformResourceChecks ->
                                           (LGDL5
                                            &
                                            (TEMP"L.CyclePosition =
                                                AttachDevices ->
                                              (LGDL6
                                               &
                                               LGDL7))))
                                   <> L.CyclePosition =
                                           ReadAccessPassword =>
                                        LGDL4
                                        &
                                        (TEMP"L.CyclePosition =
                                                   PerformResourceChecks ->
                                          (LGDL5
                                           &
                                           (TEMP"L.CyclePosition =
                                                  AttachDevices ->
                                             (LGDL6
                                              &
                                              LGDL7))))
                                  <> Error)
                              <> Error))
                          <> Error)
                        <> MsgName(Text) = LineStatus =>
                             NTWK1
                        <> Error)
                    <> Error))
               <> (MsgName(Text) = LineStatus =>
                       NTWK1
                  <> MsgName(Text) = LineInfo =>
                       LGDL1
                  <> Error))
```

```
transform MsgNkcp(MsgId: MessageId,
                  Text: String,
                  Source: ProcessName)

refcond Source ~<: TrustedProcesses||NetworkProcess

effect  (E"P:PendingRequest(P<:PendingRequests &
           P.MsgId = MsgId) =>
           E"P:PendingRequest(P<:PendingRequests &
             P.MsgId = MsgId
             &
             (P.Kind = OpRequest =>
                (MsgName(Text) = ResponseToOpRequest =>
                  ProcessedResponse(P, Text, Source)
                <> Error)
             <> P.Kind<:S"(NewVM,ConnectVM,NewUser,NewOrConnectedVM) =>
                (E"L:LineEntry(L<:Lines &
                   L.Laddr = Laddr(Text)) =>
                   LGDL6
                <> Error)
             <> P.Kind = RelinquishDevice =>
                (MsgName = DetachDevice =>
                   NKCP6
                <> Error)
             <> Error))
        <> (MsgName(Text) = Disconnect =>
              NKCP1
           <> MsgName(Text) = Logoff =>
              NKCP2
           <> MsgName(Text) = DropUser =>
              NKCP3
           <> MsgName(Text) = Link =>
              NKCP4
           <> MsgName(Text) = DetachDevice =>
              NKCP5
           <> MsgName(Text) = PurgeNkcp =>
              NKCP7
           <> Error))
```

```
transform AuthDriver (InterruptType: ?,
                       InterruptSubType:?,
                       MsgId: MessageId,
                       Text: String,
                       Source: ProcessName)

effect  (InterruptType = ExternalInterrupt =>
           (InterruptSubType = Message =>
                MessageReceived
                &
                (Source = OpProcess =>
                     MsgOp(MsgId, Text, Source)
                <> Source = URProcess =>
                     MsgUR(MsgId, Text, Source)
                <> Source = NetworkProcess =>
                     MsgNet(MsgId, Text, Source)
                <> Source ~<: TrustedProcesses||NetworkProcess =>
                     MsgNkcp(MsgId, Text, Source)
                <>Error)
           <> Error)
        <> Error)
        &
        KernelCalled(ReceiveInterrupts)
        &
        KernelCalled(ReleaseCPU)
end AuthProcess
```

3.4.1: Accounting Process Informal Description

This section contains the informal description of the Accounting Process of KVM/370.

Overview

The Trusted Process performing the accounting function, ACNTProcess, is one of the simpler processes. It accepts two flavors of accounting records from NKCPs, and saves them in a data base. It also accepts two operator commands that cause the old data base to be saved and a new empty one to be created.

The only complexity is the need for preserving the *-property when tallying the computer usage accounts. Each user is presented with a bill which contains no information derived from other users' accounting information. To accomplish this, the Accounting Process keeps its accounting records keyed on user id.

The formal specification also glosses over an important effect of the operator's commands. In "OP1", the "effect" section says nothing about actually saving the old data base, presumably as a spool file intended for the card punch.

NKCP Communication with the Accounting Process

There are two kinds of message that an NKCP can send to the Accounting Process. Their respective labels are:

- DeviceUse; and
- SystemResourceUse.

The first, DeviceUse, is indirectly sent to the Accounting Process by an NKCP whenever a device is detached from a VM. The accounting record describes the utilization of the device by the VM. The message comes to the Accounting Process via the Authorization Process.

The second message type, SystemResourceUse, is also indirectly sent to the Accounting Process by an NKCP via the Authorization Process. When a VM logs off (or is forced off, etc.), the controlling NKCP notifies the Authorization Process so that system tables may be updated and communication lines freed. As part of this message, the NKCP includes accounting information that the Authorization Process reflects to the Accounting Process.

The design purposely incorporates this roundabout form of message passing so that the user id (VM name) reported in the accounting record may be verified to actually exist as a VM controlled by the requesting NKCP.

Communication between the Operator and tha Accounting Process

There are two operator commands which affect the Accounting Process:

- ACNT; and
- SHUTDOWN.

The operator command ACNT has the following permissable parameters:

- user id;
- ALL; and
- PUNCH.

The first two are present in VM/370; the last is new in KVM. The first two cause one or more NKCPs to generate accounting records for users. These accounting records are sent as messages to the Authorization Process, which verifies that the user ids are indeed logged on and being controlled by the reporting NKCP. The Authorization Process then sends the messages on to the Accounting Process for recording.

The operator commands ACNT←PUNCH and SHUTDOWN are reflected by the Operator Process directly to the Accounting Process. The commands cause the current data base of accounting records to be saved and a new empty one to be started. Both have the same effect.

3.4.2: Accounting Process Formal Specification

```
module AcntProcess
type
MessageLabels = (SystemResourceUse,DeviceUse,ACNT←PUNCH,SHUTDOWN),
KernelFunction = (SendMessage),

constant
AuthProcess,OpProcess: ProcessName,
Error: boolean,
SendMessage(ProcessName): KernelFunction

transform KernelCalled(K:KernelFunction)

effect  true

   /* Parameter Functions */

constant
MsgName(String): MessageLabels,
User(String): VirtualMachineName,
NewPosting(String): String
```

```
type
Char,
String = list of Char,
VirtualMachineName,
ProcessName,
MessageId,
AccountingRecord = structure of(
           User = VirtualMachineName,
           Postings = set of String)

variable
Accounting: set of AccountingRecord

initial
Accounting = Empty

invariant
A"A1,A2:AccountingRecord (A1<:Accounting
                          &
                          A2<:Accounting ->
        (A1.User = A2.User -> A1 = A2))
```

```
Subdriver of AcntProcess,
            handling messages from AuthProcess

transform MsgAuth(MsgId: MessageId,
                  Text: String,
                  Source: ProcessName)

refcond Source = AuthProcess

effect  (MsgName(Text)<:S"(SystemResourceUse,DeviceUse) =>
             AUTH1(User(Text),NewPosting(Text))
        <> Error)
```

```
Subdriver of AcntProcess,
                handling messages from OpProcess

transform MsgOp(MsgId: MessageId,
                Text: String,
                Source: ProcessName)

refcond  Source = OpProcess

effect   (MsgName(Text)<:S"(ACNT←PUNCH,SHUTDOWN) =>
                OP1
         <> Error)
```

AUTH1: Accounting record from Nkcp via Authorization Process: System Resource Use or Device Use

```
transform AUTH1(User: VirtualMachineName,
                NewPosting: String)

refcond  true /* user id has been validated by AuthProcess
                 prior to the sending of this message */

effect  E"A:AccountingRecord(A<:Accounting ->
                 (A.User = User
                  &
                  NewPosting<:A.Postings))
```

OP1: Operator command to re-initialize the accounting data base

```
transform OP1

effect  N"Accounting = Empty
        &
        KernelCalled(SendMessage(OpProcess))
```

```
transform AcntDriver(InterruptType:
                InterruptSubType:
                MsgId: MessageId,
                Text: String,
                Source: ProcessName)

effect  (InterruptType = ExternalInterrupt =>
                (InterruptSubType = Message =>
                        (Source = OpProcess =>
                                MsgOp(MsgId,Text,Source)
                        <> Source = AuthProcess =>
                                MsgAuth(MsgId,Text,Source)
                        <> Error)
                <> Error)
        <> Error)
end AcntProcess
```

3.5.1: Updater Process
Informal Description

This section contains the informal description of the Updater Process of KVM/370.

Overview

The Updater creates and updates the set of databases which are collectively known as the Directory. The Directory contains information about system users, devices, terminals, and system-owned volumes.

Each user id must be unique within the system. Associated with the user id is a clearance (maximum security-level) and a list of capabilities as well as notes for the use of any NKCP to which the user may be attached.

Each real device and terminal is associated with a security-level which is the maximum level of information that may appear on that device.

KVM extends the concept of system-owned volumes to include not only the system residence volume and those having page/spool space, but also any shared real volume (i.e., any volume containing minidisks). Such a volume must be partitioned in such a way as to prevent overlap.

THE UPDATER REQUESTS

The System Security Officer or other user of the Updater may make the following types of requests:

1. Add, change or delete a user.
   The following are associated with each user:
   - user id (must be unique)
   - password
   - clearance
   - objects (a list of [object-name, access-type]) where object-name may name a minidisk or a real device.
   - notes for use by NKCP
     - privilege classes (zero or more of A..H)
     - maximum and default storage sizes
     - virtual devices (a list of [vaddr, devtype, corresponding real device]) (if any)
     - initial scheduling priority
     - initial line editing characters
     - accounting information and output distribution code
     - options (zero or more of (ECMODE, REALTIMER, ACCT, SVCOFF, BMK))
     - auto IPL system, if any

When a new user is added to the directory, all the fields must be specified, though some may be defaulted to null (objects, privileges, virtual devices and options) or to standard values (priority = 50, line- edit = [ON,'@','[','#','"']). The user id must be distinct from any user id presently in the directory.

When updating a user entry, any of the above fields may be changed. If the user id is changed, the new value must not be the same as any other user id in the directory. The following are recognized as special cases:

(a) add or delete a category in clearance
(b) add or delete an object
(c) add or delete a privilege to the privilege classes
(d) add, delete or change a virtual device.
(e) the options are not treated as a list. Rather, each is handled as a separate boolean.

Each real device mentioned in the virtual device list must have a corresponding real device mentioned in the links or dedicates section. Each real device and link should have a corresponding entry in the virtual devices, though a failure to do so is not an error and the entry may be stored after confirmation by the System Security Officer (SSO).

2. Update the security lattice
   Add a special-access compartment

3. Create, change and delete devices and terminals (lines).
   Each device has the following associated with it:
   - Device address and type
   - clearance (maximum security level)
   - minimum security level, if any
   - flags:
     - trusted/untrusted device
     - system-owned (i.e. owned volumes and only owned volumes may be mounted).

4. Create, format and allocate real System Owned Volumes

   a. Define volume
      - volume serial number (must be unique)
      - maximum security-level

   b. Label volume (volume must be mounted and attached to Updater)
      - volume serial number
      - real device

sec-level(device) must dominate sec-level(volume) existing volume label(if any) (cyl 0, track 0, record 3) is typed out for SSO who must confirm that he wants to format the volume.

A label and blank allocation record (all cylinders except 0 available) are written on the volume, which is then detached from the Updater and attached to "SYSTEM".

THE FOLLOWING REQUIRE THE VOLUME BE MOUNTED AND ATTACHED TO "SYSTEM"

c. Delete volume. The volume is deleted from the owned volume list. All cylinders marked as classified have home addresses written on each track (which clears the rest of the track) and are marked available.

d. Allocate space
  - type of space:

    (Global areas - security level need not be specified - always system-high - volume must be system-high)

      PAGE (suballocated by DASD page allocator)
      SPOOL (cylinders suballocated by Spool File Memory, records by NKCPs)
      Directory
      IPL area

    (areas used by NKCPs - security level must be specified)

      WarmStart or checkpoint area
      Shared segment area
        - name of segment (must be unique)
      Temporary Disk Space (suballocatd by NKCP)
      Minidisks
        - name (must be unique)
        - passwords or ACLs
          (ACLs are array by accesstype of [user id])
  - security-level (except global areas which are always system high. The security level of an area must be dominated by the clearance of the volume.)
  - starting cylinder
  - number of cylinders

The requested area must not overlap any area currently allocated on the volume. Thus, to increase the size of an area, the area must be deleted and then re-allocated in the increased area.

When a shared segment area or Minidisk is created, the name must be different from any other object of the same kind in the system.

If passwords are used, 3 passwords are provided for each minidisk: [read password, write password, simultaneous-write password].

If ACLs are used, the following functions are to be provided:

- Create a "group" ACL
- Add or delete a user id to an ACL
- Add or delete an ACL to another ACL

and the following kinds of ACLs are maintained:

- Read access (may read only)
- Write Access (may write, if no other user is using the disk)
- Multiple Access (may write, even though other users are using the disk)
- Control Access (may modify the ACL for this disk)

If audit trails are kept, the first cylinder of each minidisk will be set aside for audit trails and, possibly, ACLs. Hence, the virtual size of the minidisk will be one cylinder smaller than that specified by the SSO.

e. Free space
- type
- name

When an existing area is freed, the contents should, in principle, be erased unless the area is being reassigned to the same security level. In addition, no further accesses to the deleted area should be allowed. This latter consideration brings up the problem of capability revocation, see below.

f. Delete space - like 'Free Space' except that the space is made permanently unavailable for allocation and hence need not be cleared until the volume is deleted.

REVOCATION OF CAPABILITIES

One of the problems encountered in capability-based systems is that of revoking a capability after it has been granted and used. When a process has been granted a capability it may have multiple links to the control blocks which represent that capability; in some systems the original grantee may be able, in turn, to grant the capability or some subset to other processes. Further, there may be other capabilities that depend on the original capability to be meaningful. Deleting the original capability without deleting those dependent on it may produce a potential security violation.

There is a denial of service problem which further complicates matters. If the Security Kernel revokes a capability without notice, the process may be unable to continue working. Moving an empty chair won't bother anyone - unless she is starting to sit down in it.

A number of approaches to this problem have been used in KVM. The most general approach is an extension of that used in VM/370 - currently granted capabilities are never revoked. If, while a user is logged on, a new directory is created in which his privileges are changed, the changes do not take effect until he logs off (thus releasing his current capabilites) and logs back on. If the changed privilege is access to a minidisk, he can also get the new privilege by detaching the virtual device (thus releasing the capability) and reLINKing to it (thus getting the new capability). Similarly, in KVM the Initiator will be consulted only when a user logs on to the system or attempts to link to a disk or have a device attached. As long as he is using it, changes to the security database (the Directory) will not affect him.

Another approach is to refuse to delete capabilities unless all dependent capabilties have also been deleted. For example, the Kernel will execute the Destroy VM call only if all page frames, page slots and address spaces belonging to the subject VM have first been detached or destroyed. Similarly, an NKCP (process) can be destroyed only if it has no outstanding requests which can cause completion signals to be sent to the non-existent process. A page frame may not be released by a process if the process has pending I/O involving that page frame.

Another approach involves the management of equivalent capabilities. In a paging system, for example, all page frames and page slots are equivalent to each other outside of the paging management module. Thus, the Kernel can safely move virtual pages between main memory and DASD without notice to the processes. The Kernel must, however, respect certain critical regions within NKCPs when the status of a page is expected to remain constant. This problem is discussed more completely in other papers.

The approach used in managing the Directory is a mixture of several. Thus, user ids and their access rights can be freely changed in the Directory without interacting with current use of those rights. The changes will take effect when the user releases his current use of the capability. Allocation of space on disk volumes is a slightly different problem, however. If a minidisk is deleted and the space reassigned while a process is using it, another process may get access to the reassigned space, resulting in two processes with read-write access to the same disk area - a clear violation of KVM's security policy.

To avoid such a violation, we forbid deletion of DASD space while a process is using it. If the SSO requests that an area be freed or deleted, the Updater will send a message to the Initator requesting permission to do so. The Initiator will reply OK if the area is not currently in use, otherwise NO. In either case, the Initiator will mark the area as unavailable for use so that no process will be granted use of it while the SSO is updating the directory. When the SSO finishes updating the directory, a new copy is created and the Updater notifies the Initator to use the new copy for all future requests. Upon receipt of that message, the Initiator will read in the new directory, which will result in all 'unavailable' marks being released.

Finally, in order to allow the SSO to conveniently do his work in the face of the possibility of having a request denied because the named area is in use, we add the following requests:

5. Create and replace directories

a. Create new directory - when the SSO is satisfied with the changes he has made, he can cause the new directory to be built and installed. If he has changed the security level of an area, he will be required to confirm his desire to erase, up- or downgrade the contents of that area.

b. Reuse old directory - when the SSO has begun a series of directory changes but is unable to complete them due to denial of a request or a typing error, he can request that the old directory be re-installed. This sends a message to the Initiator causing it to forget all its 'not available' marks and continue using the old directory.

c. Start over - if the SSO has a problem with the series of directory changes he is making,. but doesn't want to release the 'not available' marks on his relation requests, he can request the Updater to start over with a fresh copy of the old directory for him to update, in effect 'forgetting' all changes he has made while preventing any access to areas he wants to reassign.

Note that if the area is in use, the SSO's request will be denied, but 'not available' flags will still be set, so that the area will eventually, as processes log off, be available for freeing.

3.5.2: Updater Process
Formal Specification

```
module UpdaterProcess
type
Char,
String = list of Char,

DeviceAddress,
LineAddress,
VolumeId,
ProcessName,
VirtualMachineName,
DeviceTypes = (Reader,Printer,Punch,TapeDrive,NonsharableDisk)

constant
Dominates(ProcessName,ProcessName): boolean,
DeviceType(DeviceAddress): DeviceTypes,
#MaxCylinders: integer,
#Cylinders(VolumeId): integer
```

```
type
AccessModes = (Read,Write),

PossibleEntries = (Paging,Spooling,MiniDisk,Unknown,System),

CylInt = I"I:integer(1 <= I & I <= #MaxCylinders),

DirectoryEntry = structure of (
          UserId = VirtualMachineName,
          LogonPassword = String,
          DialPassword = String,
          LinkPassword = String,
          MaxSecLevel = ProcessName,
          MinSecLevel = ProcessName,
          DedicatedDevices = set of DedicatedDeviceEntry,
          Links = set of MDLinkEntry,
          IplDefined = boolean,
          AccessPasswords = set of AccessPasswordEntry),

LineEntry = structure of (
          Laddr = LineAddress,
          MinSecLevel = ProcessName,
          MaxSecLevel = ProcessName),

AccessPasswordEntry = structure of (
          SecLevel = ProcessName,
          Password = String),

DedicatedDeviceEntry = structure of (
          Raddr = DeviceAddress,
          VolSecLevel = ProcessName,
          Access = set of AccessModes),

MDLinkEntry = structure of (
          MDName = MiniDiskName,
          Access = set of AccessModes),

URPOwnedDeviceEntry = structure of (
          Raddr = DeviceAddress,
          MaxSecLevel = ProcessName,
          MinSecLevel = ProcessName),

NonsharableDriveEntry = structure of (
          Raddr = DeviceAddress,
          MaxSecLevel = ProcessName,
          MinSecLevel = ProcessName),

SharableDriveEntry = structure of (
          Raddr = DeviceAddress,
          SecLevel = ProcessName),
SharedVolumeEntry = structure of (
        Volume = VolumeId,
        SecLevel = ProcessName,
        Map = set of CylMap),

CylMap = structure of (
        Cylinders = CylInt >< CylInt,
        Category = PossibleEntries),

MiniDiskEntry = structure of (
        MDName = MiniDiskName,
        ContainingVolume = VolumeId,
        Cylinders = CylInt >< CylInt,
        SecLevel = ProcessName,
        AccessControlList = set of ACLEntry),

ACLEntry = structure of (
        User = VirtualMachineName,
        Access = set of AccessModes)
```

```
variable
URPOwnedDevices: set of URPOwnedDeviceEntry,
NonsharableDrives: set of NonsharableDriveEntry,
SharableDrives: set of SharableDriveEntry,
SharedVolumes: set of SharedVolumeEntry,
MiniDisks: set of MiniDiskEntry,
Lines: set of LineEntry,
UserDirectory: set of DirectoryEntry
```

```
transform Updater Process

effect  DistinctDeviceAddresses
        &
        LegalUserDirectory
        &
        LegalLines
        &
        LegalMiniDisks
        &
        LegalSharedVolumes
        &
        LegalSharableDrives
        &
        LegalNonsharableDrives
        &
        LegalURPOwnedDevices
```

```
DistinctDeviceAddresses =

        A"U:URPOwnedDeviceEntry(U<:URPOwnedDevices ->
            (A"NS:NonsharableDriveEntry(NS<:NonsharableDrives ->
                (U.Raddr ~= NS.Raddr))
             &
             A"S:SharableDriveEntry(S<:SharableDrives ->
                (U.Raddr ~= S.Raddr))))
        &
        A"NS:NonsharableDriveEntry(NS<:NonsharableDrives ->
             A"S:SharableDriveEntry(S<:SharableDrives ->
                 NS.Raddr ~= S.Raddr))
```

```
LegalUserDirectory =

        A"U1,U2:DirectoryEntry(U1<:UserDirectory
                               &
                               U2<:UserDirectory ->
           (U1.UserId = U2.UserId -> U1 = U2))
        &
        A"U:DirectoryEntry(U<:UserDirectory ->
           (Dominates(U.MaxSecLevel,U.MinSecLevel)
            &
            LegalDedicatedDevices(U)
            &
            LegalLinks(U)
            &
            LegalAccessPasswords(U)))
```

```
LegalDedicatedDevices(U:DirectoryEntry) =

        A"E1,E2:DedicatedDeviceEntry(E1<:U.DedicatedDevices
                                     &
                                     E2<:U.DedicatedDevices ->
           (E1.Raddr = E2.Raddr -> E1 = E2))
        &
        A"E:DedicatedDeviceEntry(E<:U.DedicatedDevices ->
            E"D:URPOwnedDeviceEntry(D<:URPOwnedDevices &
               (D.Raddr = E.Raddr
                &
                (DeviceType(E.Raddr) = Reader ->
                       (E.VolSecLevel = nil
                        &
                        E.Access = S"(Read)))
                &
                (DeviceType(E.Raddr)<:S"(Printer, Punch) ->
                       (E.VolSecLevel = nil
                        &
                        E.Access = S"(Write)))
                &
                (DeviceType(Raddr) = TapeDrive ->
                       (Dominates(D.MaxSecLevel,E.VolSecLevel)
                        &
                        Dominates(E.VolSecLevel,D.MinSecLevel)
                        &
                        Dominates(U.MaxSecLevel,E.VolSecLevel)
                        &
                        ~Empty(E.Access)))
               ))
            xor
            E"D:NonsharableDriveEntry(D<:NonsharableDrives &
               (D.Raddr = E.Raddr
                &
                Dominates(D.MaxSecLevel,E.VolSecLevel)
                &
                Dominates(E.VolSecLevel,D.MinSecLevel)
                &
                Dominates(U.MaxSecLevel,E.VolSecLevel)
                &
                ~Empty(E.Access))))
        &
```

```
A"L1,L2:MDLinkEntry(L1<:U.Links
                    &
                    L2<:U.Links ->
   (L1.MDName = L2.MDName -> L1 = L2))
&
A"L:MDLinkEntry(L<:U.Links ->
    E"M:MiniDiskEntry(M<:MiniDisks &
       (M.MDName = L.MDName
        &
        E"A:ACLEntry(A<:M.AccessControlList &
           (A.User = U.UserId
            &
            A"AM:AccessModes(AM<:L.Access ->
                AM<:A.Access)))
        &
        Dominates(U.MaxSecLevel,M.SecLevel)))
&
~Empty(L.Access))
```

```
LegalAccessPasswords(U:DirectoryEntry) =

        A"A1,A2:AccessPasswordEntry(A1<:U.AccessPasswords
                                    &
                                    A2<:U.AccessPasswords ->
           (A1.SecLevel = A2.SecLevel -> A1 = A2))
        &
        A"A:AccessPasswordEntry(A<:U.AccessPasswords ->
           (Dominates(U.MaxSecLevel,A.SecLevel)
            &
            Dominates(A.SecLevel,U.MinSecLevel)))
```

```
LegalLines =

        A"L1,L2:LineEntry(L1<:Lines
                               &
                               L2<:Lines ->
           (L1.Laddr = L2.Laddr -> L1 = L2))
        &
        A"L:LineEntry(L<:Lines ->
            Dominates(L.MaxSecLevel,L.MinSecLevel))
```

```
LegalMiniDisks =

        A"M1,M2:MiniDiskEntry(M1<:MiniDisks
                                &
                              M2<:MiniDisks ->
           (M1.MDName = M2.MDName -> M1 = M2))
        &
        A"M:MiniDiskEntry(M<:MiniDisks ->
           (LegalContainingVolume(M)
            &
            M.Cylinders.1 < M.Cylinders.2
            &
            M.Cylinders.2 <=#Cylinders(M.ContainingVolume)
            &
            M.Cylinders.1 < #Cylinders(M.ContainingVolume)
            &
            LegalAccessControlList(M)))
```

```
LegalContainingVolume(M:MiniDiskEntry) =
        E"S:SharedVolumeEntry(S<:SharedVolumes &
           (S.Volume = M.ContainingVolume
           &
           Dominates(S.SecLevel,M.SecLevel)))
```

```
LegalAccessControlList(M:MiniDiskEntry) =

        A"A1,A2:ACLEntry(A1<:M.AccessControlList
                                  &
                                  A2<:M.AccessControlList ->
            (A1.User = A2.User -> A1 = A2))
        &
        A"A:ACLEntry(A<:M.AccessControlList ->
            (E"D:DirectoryEntry(D<:UserDirectory &
                (D.UserId = A.User))
             &
             ~Empty(A.Access)))
```

```
LegalSharedVolumes =

        A"S1,S2:SharedVolumeEntry(S1<:SharedVolumes
                                 &
                                 S2<:SharedVolumes ->
           (S1.Volume = S2.Volume -> S1 = S2))
        &
        A"S:SharedVolumeEntry(S<:SharedVolumes ->
           (LegalMap(S)))
```

```
LegalMap(S:SharedVolumeEntry) =

          /* non-overlap */
          A"M1,M2:CylMap(M1<:S.Map
                               &
                               M2<:S.Map ->
              (M1.Cylinders.1 > M2.Cylinders.2
               |
               M1.Cylinders.2 < M2.Cylinders.1))
          &
          A"M:CylMap(M<:S.Map ->
              (/* each entry non-empty */
               M.Cylinders.2 > M.Cylinders.1
               &
               /* no cylinders unaccounted for */
               M.Cylinders.2 ~= #Cylinders(S.Volume) ->
                    E"M1:CylMap(M1<:S.Maps &
                         M1.Cylinders.1 = M.Cylinders.2 + 1)))
          &
          E"M:CylMap(M<:S.Map &
              (M.Cylinders.1 = 1))
          &
          /* each MiniDisk actually logged */
          A"M:CylMap(M<:S.Map ->
              (M.Category = MiniDisk ->
                    E"MD:MiniDiskEntry(MD<:MiniDisks &
                        (MD.ContainingVolume = S.Volume
                         &
                         MD.Cylinders = M.Cylinders))))
```

```
LegalSharableDrives =

          A"SD1,SD2:SharableDriveEntry(SD1<:SharableDrives
                                       &
                                       SD2<:SharableDrives ->
             (SD1.Raddr = SD2.Raddr -> SD1 = SD2))
```

```
LegalNonsharableDrives =

          A"NS1,NS2:NonsharableDriveEntry(NS1<:NonsharableDrives
                                          &
                                          NS2<:NonsharableDrives ->
             (NS1.Raddr = NS2.Raddr -> NS1 = NS2))
          &
          A"NS:NonsharableDriveEntry(NS<:NonsharableDrives ->
               Dominates(NS.MaxSecLevel,NS.MinSecLevel))
```

```
LegalURPOwnedDevices =

          A"U1,U2:URPOwnedDeviceEntry(U1<:URPOwnedDevices
                                      &
                                      U2<:URPOwnedDevices ->
              (U1.Raddr = U2.Raddr -> U1 = U2))
          &
          A"U:URPOwnedDeviceEntry(U<:URPOwnedDevices ->
              (Dominates(U.MaxSecLevel,U.MinSecLevel)
               &
               DeviceType(Raddr)<:S"(Reader,Printer,Punch,TapeDrive)))

 end UpdaterProcess
```